Scalable Synchronization and Scheduling of Distributable Real-Time Threads

Sherif F. Fahmy

Preliminary examination proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

December 9, 2008
Blacksburg, Virginia

Keywords: Distributed Real-Time Scheduling, Software Transactional Memory,
Time/Utility Functions, Utility Accrual Scheduling

Scalable Synchronization and Scheduling of Distributable Real-Time Threads

Sherif F. Fahmy

(ABSTRACT)

Advances in computer architecture and networking have increased the number of distributed real-time systems being developed. At the same time, chip manufacturing considerations have caused a paradigm change in the computer industry: multicore and hyperthreading architectures, rather than increasing clock rates, are now the preferred method for increasing computing performance. The twin occurrences necessitate a new shift in focus of the real-time community to distributed and multiprocessor solutions.

In this dissertation proposal, we investigate a number of issues associated with these emerging architectures. First, to address the scheduling problem on distributed systems, we investigate the relative merits of collaborative scheduling, an approach to system wide scheduling in which all nodes participate in a collaborative manner. Towards that end, we design three different collaborative scheduling algorithms (ACUA, QBUA and DQBUA). We analytically and empirically evaluate the properties of these algorithms. The result of our studies indicate that collaborative scheduling algorithms can provide better timeliness assurances compared to non-collaborative scheduling algorithms (where nodes do not cooperate while constructing their schedules).

However, our studies also indicate that collaborative scheduling algorithms, particularly when distributed dependencies are involved, can have a significant overhead. There are a class of applications, such as Network Centric Warfare [12], that have sufficiently large execution timescales to benefit from the improved timeliness of collaborative scheduling. We also believe that collaborative scheduling, because it integrates failure detection with the actual scheduling algorithm, can provide more seamless performance assurances during failures than previous attempts where failure detection (and the response to such failure) was disjoint from the actual scheduling algorithm.

We identify distributed dependencies as one of the major sources of overhead in collaborative scheduling algorithms. Particularly, the cost of distributed lock management and distributed deadlock detection and resolution can become quite significant. This is particularly important for emerging multicore architectures where concurrency is becoming the norm rather than the exception. In order to alleviate this problem, we propose an alternative to lock-based concurrency control known as software transactional memory (or STM).

In this proposal, we indicate the set of algorithms and protocols that are necessary for making STM a part of a real-time programmer's repertoire. In addition, we design two different schedulability analysis algorithms for computing the response times of threads programmed using STM as the concurrency control mechanism. The first of these algorithms targets distributed systems where each node is a uniprocessor, while the second targets distributed systems where each node is a multiprocessor. We also identify a number of research problems associated with STM that we propose to tackle after the preliminary exam.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Advances in the computer, and networking, industry have resulted in the increased use of distributed systems. The real-time domain has also seen a rise in the number of applications deployed on distributed systems. These applications range from relatively tightly coupled systems, e.g., embedded distributed systems used to control, for example, cars and planes, to applications deployed on loosely coupled systems such as those envisioned by the DoD for its vision of Network Centric Warfare [12].

Thus, it becomes necessary to extend the standard analysis techniques of classical real-time theory to deal with such systems. Some of these emerging networked embedded systems are dynamic in the sense that they operate in environments with uncertain properties (e.g., [12]). These uncertainties include transient and sustained resource overloads (due to context-dependent activity execution times), arbitrary activity arrivals and completions, and arbitrary node failures and message losses. Reasoning about *end-to-end* timeliness is a difficult and unsolved problem in such systems. Another distinguishing feature of such systems is their relatively long activity execution time scales (e.g., milliseconds to minutes), which permits more time-costly real-time resource management.

In particular, it is necessary to provide some measure of assurances on the end-to-end timeliness of the computations on such system if they are to support real-time applications. The problem of providing end-to-end timeliness assurances for distributed real-time systems has been studied in the past (e.g., [20, 22, 74, 75]). These past efforts can be broadly categorized into two classes: *independent node scheduling* and *collaborative scheduling*.

In the independent scheduling approach (e.g., [20, 75]), threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes. Fault-management is separately addressed by *thread integrity protocols* that run concurrent to thread execution. Thread integrity protocols employ failure detectors (FDs) to detect failures of the thread abstraction, and to deliver failure-exception notifications [20]. In the collaborative scheduling approach (e.g., [74]), nodes explicitly cooperate to construct system-wide thread schedules, anticipating and detecting node failures using FDs.

FDs employed in both paradigms in past efforts have assumed a totally synchronous computational model—e.g., deterministically bounded message delay. While the synchronous model is easily adapted for real-time applications due to the presence of a notion of time, this results in systems with low coverage—i.e., the high likelihood for the resulting timing assurances to be violated, when the synchrony assumptions are violated at run-time (e.g., due to overloads, or other exigencies). On the other hand, it is difficult to design real-time algorithms for the asynchronous model due to its total disregard for timing assumptions.

Thus, there have been several recent attempts to reconcile these extremes. For example, in [2], Aguilera *et. al.* describe the design of a fast FD for synchronous systems and show how it can be used to solve the consensus problem for real-time systems. Their algorithm achieves the optimal bound for both message and time complexities. In [44], Hermant and Widder describe the *Theta*-model, where only the ratio, $\mathbf{\Theta}$, between the fastest and slowest message is known. This increases the coverage of algorithms designed under this model, as less assumptions are made about the system. Though $\mathbf{\Theta}$ is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties. In this dissertation proposal, we attempt to provide timeliness assurances for systems where propagation delay and message loss can be stochastically described as well as for standard synchronous systems.

To summarize, there are different approaches to providing end-to-end timeliness. These approaches can be either collaborative or independent and can be deployed on systems with different levels of "synchronicity". One of the main questions we try to answer in this proposal is whether or not the higher overhead of collaborative scheduling is justifiable. Another important research point addressed in this proposal, is whether the integration of failure detection and handling into a scheduling solution offers any advantages over using orthogonal failure handling techniques.

One of the major contributors to the overhead of collaborative scheduling is the cost of distributed concurrency control. In this proposal, we investigate alternatives to the traditional method of using locks and condition variables to reduce this overhead. In particular, we investigate the use of software transactional memory (STM), and study the real-time assurances that can be provided when STM is used to manage concurrency control.

To summarize, the aim of this dissertation is to study the problem of scheduling distributed threads. In order to do so, we investigate the assurances that can be provided on partially synchronous distributed systems, study the performance of collaborative scheduling and compare it with independent node scheduling, identify performance bottlenecks in scheduling distributed systems and attempt to remedy them.

The rest of this Chapter is organized as follows, in Section 1.1, we describe the timeliness models used in this dissertation proposal. In particular, since some of our target systems are soft real-time systems that are subject to overloads, we describe the Time Utility Function

(TUF) timeliness model, and show how TUFs can be used to describe both the urgency and importance of a task. Using TUFs allows us to develop real-time scheduling algorithms that exhibit graceful performance degradation during overloads.

In Section 1.2, we describe our current research contribution. In Section 1.3, we present an overview of the work we plan to conduct after the preliminary exam in order to achieve the goals of this dissertation and Section 1.4 provides a road map for the rest of the proposal.

1.1 Timeliness models

A real-time system can be either hard real-time or soft real-time. In hard real-time systems, all tasks must meet their deadlines. This timeliness model is essential when the consequences of not meeting a deadline could be disastrous (e.g., in the code controlling a nuclear reactor). For hard real-time systems, a deadline for each task is sufficient to describe the timeliness requirements of the application.

On the other hand, in a soft real-time system, if a task does not meet its deadline, the consequences are not disastrous. In such a system, it is possible to tolerate periods of overload (when the offered load of the application is more than the processor can service) by missing deadlines. In such a scenario, it becomes important to make a decision about which set of tasks will be allowed to meet their deadline and which set of tasks will be delayed.

The notion of "importance" plays an important role in these scenarios. If each computation task has an importance, it is in the best interest of the application to meet the deadline of tasks with higher importance and delay less important tasks.

The urgency, closeness of a deadline, of an activity is sometimes orthogonal to the relative importance of the activity – e.g., the most urgent activity may be the least important, and vice versa; the most urgent may be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is desirable. Thus, a distinction has to be made between urgency and importance during overloads. (During underloads, if all time constraints are deadlines, optimal algorithms exist that can meet all deadlines – e.g., EDF [21].)

Deadlines cannot express both urgency and importance. Thus, our soft timeliness criteria considers the time/utility function (or TUF) timeliness model [48] that specifies the utility of completing an activity as a function of that activity's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF; Figure 1.1 shows some examples. An activity's TUF decouples its importance and urgency – urgency is measured on the X-axis, and importance is denoted (by utility) on the Y-axis. Some real-time systems also have activities with non-deadline time constraints, such as those where the utility attained for activity completion varies (e.g., decreases, increases) with completion time.

When activity time constraints are expressed with TUFs, the scheduling optimality criteria

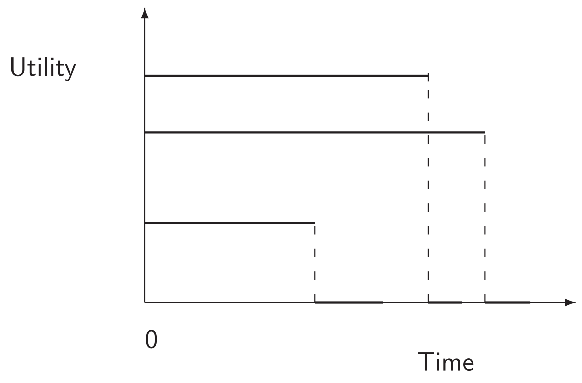

Figure 1.1: Step TUFs

are based on maximizing accrued activity utility – e.g., maximizing the total activity accrued utility. Such criteria are called utility accrual (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms. UA algorithms that maximize total utility under downward step TUFs (e.g., [21]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important activities over less important ones (since more utility can be attained from the former), irrespective of activity urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [21] in the sense that the algorithms strive their best to feasibly complete as many high importance activities – as specified by the application through TUFs – as possible. Consequently, high importance activities that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note also that EDF's optimal timeliness behavior is a special-case of UA scheduling.

1.2 Summary of Current Research and Contributions

The main aims of the research being conducted are; 1) Determine the possibility of providing timeliness assurances in non-synchronous systems that are subject to failures, 2) Investigate the properties of collaborative scheduling, 3) Identify the major performance bottlenecks in collaborative scheduling, and 4) Attempt to overcome these bottlenecks.

Towards that end we have conducted a series of studies to address these problems. Specifically, in [33] (described in Chapter 3), we attempt to address the first problem by solving the consensus problem on a partially synchronous system, and then using the consensus algorithm as a basis for collaborative scheduling.

The partially synchronous system we target is one where both communication delay and communication failure are described stochastically [17]. On top of this system we design an $\mathcal{S}$-class failure detector and use that to develop a consensus algorithm. We then use the consensus algorithm as the basis for a collaborative scheduling algorithm, ACUA. Essentially, this addresses two issues. First, we show how it is possible to provide stochastic timeliness guarantees on partially synchronous systems. Second, we analytically and empirically prove that the collaborative scheduling algorithm thus developed can provide better timeliness, when compared to independent scheduling, for systems that can tolerate its higher overhead.

We then attempt to reduce the communication overhead of the algorithm by using a quorum-based approach, QBUA [30] (described in Chapter 4), instead of the consensus based approach used in ACUA. The algorithm developed uses the partially synchronous model and QoS failure detectors described in [17]. Again, we analytically and empirically compare the algorithm developed to other algorithms, both independent and collaborative. These first two algorithms, ACUA and QBUA, did not consider distributed dependencies. Thus we developed a modified quorum-based algorithm, DQBUA [32], specifically to address this issue. DQBUA is described in Chapter 5.

While our research indicates that collaborative scheduling can provide better timeliness by taking global scheduling information into account, it also indicates that collaborative scheduling has a high overhead that may not be suitable for all situations.

Thus, our next step was to identify the largest contributor to this overhead. Naturally, the communication overhead has a significant effect on the overhead of the algorithm, but this, apart from network techniques to improve communication efficiency and algorithmic designs to reduce communication, was relatively inevitable. One interesting source of overhead, was the concurrency control portion of the scheduling algorithms.

The standard method of using locks for concurrency control introduced a significantly large overhead to the scheduling algorithm. This overhead manifested itself in both the design of the sequencing algorithm used to arrange the tasks in an order that did not violate the precedence constraints imposed on them by their locks, and in detecting and resolving deadlock when it occurs.

The problem becomes more complicated when we consider distributed dependencies which can result in distributed deadlocks. The standard methods for solving such problems in distributed systems [11, 19, 24, 25, 27, 56, 57, 65, 76], break down for real-times systems [81]. Essentially, this occurs because deadlock is no longer a stable property when deadlines are considered. In normal distributed systems, deadlocks are a stable property, and algorithms, such as edge chasing deadlock detection algorithms, depend on this property for their correctness. However, the spontaneous termination of tasks, when their deadline passes, in real-time systems invalidates the assumption that deadlocks are stable and necessitates a large number of trade-offs in designing real-time distributed deadlock detection and resolution algorithms [81].

Thus we came to the conclusion that seeking alternatives to lock-based concurrency control can offer significant performance and semantic improvements. For example, it is possible to come up with non-lock based concurrency control solutions that are not subject to deadlocks. This is beneficial from two points of view. First, it reduces the complexity of the scheduling algorithm in terms of both its semantic complexity (by freeing it from dealing with deadlocks) and time complexity (by eliminating the need for deadlock detection and resolution, one of the major contributors to scheduling time complexity). Second, the use of non-lock based concurrency control can significantly increase the semantic simplicity of the application code being written thus improving programmer productivity. After considering several alternatives to lock based concurrency control, some of which are briefly overviewed in Section 2.0.1, we concluded that software transactional memory would be most suited for our purpose.

In order to incorporate STM into distributed real-time systems, we have developed several response time analysis techniques to provide timeliness assurances for such systems. First, in Chapter 7, we provide such an analysis for distributed systems where each node is a single processor, and then extend this notion to include distributed systems, in Chapter 8, where each node is a multiprocessor. In Section 1.3 we indicate the post preliminary examination work we intend to pursue in order to achieve our goal of making STM one of the tools available for real-time programmers in an attempt to reduce the overhead introduced by lock-based concurrency control.

The following list briefly summarizes the contribution of the proposal:

1. Designed a consensus-based collaborative scheduling algorithm, ACUA, that can provide fault tolerant timeliness assurances in partially synchronous systems.
2. Provided empirical and analytical evaluation of the properties of ACUA.
3. Designed a quorum-based collaborative scheduling algorithm, QBUA, designed for fault tolerant partially synchronous systems in an attempt to reduce the communication overhead of the consensus-based algorithm, ACUA.
4. Provided empirical and analytical evaluation of the properties of QBUA.
5. Designed a modified version of QBUA, DQBUA, that can handle distributed dependencies.
6. Provided empirical and analytical evaluation of the properties of DQBUA.
7. Identified the problems associated with lock-based scheduling in our target domain and suggested STM as an alternative.
8. Designed a schedulability analysis algorithm for distributed real-time systems where each node is a uniprocessor and concurrency control is managed using STM.
9. Designed a schedulability analysis algorithm for distributed real-time systems where each node is a multiprocessor and concurrency control is managed using STM.

1.3 Summary of Proposed Post Preliminary-Exam Work

Based on our current direction of research, we propose the following work:

- **Investigate Different Progress Guarantees for Real-Time STM**: Since STM is, essentially, a non-lock based method for concurrency control, it can only provide non-blocking progress guarantees. The three main categories of non-blocking progress guarantees, in descending order of strength, are; 1) Wait-freedom: this combines system wide throughput guarantees with starvation-freedom, 2) Lock-freedom: this provides only guarantees on system-wide throughput but may allow individual threads to starve, and 3) Obstruction-freedom: this is the weakest non-blocking progress guarantee and ensures that a single thread executed in isolation for a bounded number of steps will complete its operation.

  One of the trends in software transactional memory is to implement an obstruction-free STM system (e.g., [41]), and to leave stronger non-blocking guarantees of progress to an orthogonal contention manager. Several contention managers have been proposed and studied in the literature (e.g., [79,91]). However, none of these contention managers are designed with real-time constraints in mind. We propose to design a set of real-time contention managers that take real-time scheduling criteria into account. The effect of these contention managers will be both empirically and analytically evaluated to see if they offer any performance assurances for real-time systems. Another trend is to drop the guarantee of even obstruction-freedom [28] in order to speed up execution. We will study such an approach and determine its viability for real-time systems.

- **Design Distributed Cache Coherence Protocols for real-time STM**: The use of RPCs, and other forms of remote procedure invocations, to design distributed systems has been extensively investigated. An alternative that has seen little research is that of using a distributed cache coherence protocol and thus turning distributed applications into a data migration paradigm.

  We intend to design a distributed cache coherence protocol that can provide some real-time assurances. On top of this distributed cache coherence protocol, we can design an STM system and investigate it properties. The contribution here is two-fold, first we propose to design a distributed cache coherence protocol which can provide timeliness assurances, second, we propose to investigate the different trade-offs usually considered in designing STMs (more details can be found in Chapter 6, but this includes, for example, whether to use eager or lazy memory updates). We believe that the effect of these design parameters for distributed STM will be different from their effect on stand alone systems (for which they have been exhaustively investigated), thus we propose to qualify the effect of these parameters on the novel environment of a distributed STM.

- **Compiler Instrumentation for STM**: One of the major factors affecting the wide-spread use of STM in production software is the high overhead involved in instrumenting all loads and stores to provide the semantics of STM. Specifically, the access of shared variables outside transactions can result in a weak atomic semantics (i.e., atomicity is guaranteed among transactions, but non-transactional code accessing a shared variable can violate atomicity).

There are several possible solutions to such a problem, for example, using synchronization barriers or instrumenting all loads and stores in the code. However, as previously mentioned, this would greatly increase the overhead of the STM system. What is required is an automatic static analysis of the code that instruments only those loads and stores that need the instrumentation. This would greatly reduce the overhead of the STM system. This area has not been adequately researched, although similar compiler related research, alias analysis and escape analysis, have been considered. We believe that the nature of most real-time code lends itself to a reasonably efficient implementation of automatic compiler instrumentation of loads and stores in STM transactions. Thus we propose to design and implement different static analysis techniques for the automatic instrumentation of loads and stores in real-time systems.

- **Integrating hard and soft real-time analysis for STM**: As mentioned in Section 1.1, there are two different approaches to timeliness. Our current research on STM (see Chapters 7 and 8) concentrate on providing timeliness assurances for hard real-time systems. We proposed to extend this analysis to deal with soft real-time systems using the timeliness model described in Section 1.1.

  This will allow programmers to use STM in systems where overloads (due to, for example, context-dependent activity execution times) can occur, thus increasing the coverage of the algorithms proposed.

- **Hybrid data/code migration**: Instead of using either data or code migration to develop distributed applications, it may be possible to provide performance gains if a hybrid solution that can move either code or data, depending on some performance criteria, is used. In Chapter 6, we describe some of these criteria and discuss the various trade-offs necessary in achieving this goal. Studying the various algorithms that can be used to achieve this is also a proposed post preliminary examination research point.

1.4 Proposal outline

The rest of this dissertation proposal is organized as follows, in Chapter 2, we provide a brief review of the relevant literature. Chapter 3 presents a collaborative scheduling algorithm and shows how this can provide better timeliness assurances than independent scheduling algorithms. In Chapter 4, we attempt to reduce the overhead of the consensus-based algorithm designed in Chapter 3 by using a quorum-based approach. In Chapter 5, the quorum-based approach is extended to deal with distributed dependencies.

We present our case for using software transactional memory (or STM) in Chapter 6, provide an overview of the requirements of incorporating STM into real-time systems and propose a set of problems we can solve to make STM in distributed real-time systems a reality. In Chapters 7 and 8, we present response time analysis for uniprocessor and multiprocess distributed systems programmed using STM respectively. We conclude the proposal in Chapter 9.

Chapter 2

Past and Related Work

As mentioned in Chapter 1, the main aim of this dissertation is to study the scheduling problem for real-time distributed systems. Specifically, the main aims of the research being conducted are; 1) Determine the possibility of providing timeliness assurances in non-synchronous systems that are subject to failures, 2) Investigate the properties of collaborative scheduling, 3) Identify the major performance bottlenecks in collaborative scheduling, and 4) Attempt to overcome these bottlenecks.

There have been a number of papers published in the literature addressing fault tolerant distributed real-time scheduling. In this chapter, we provide a brief overview of the publications most relevant to our current work. As mentioned in Chapter 1, there are two main approaches to scheduling threads on distributed real-time systems; collaborative and independent. Most past work has focused on independent scheduling due to its simplicity and low overhead.

In independent scheduling, each node in a distributed system schedules the tasks it hosts without recourse to communication with other nodes. Therefore, it was simple to extend the state of the art in single node scheduling to accommodate this model. A task making a remote invocation propagates its scheduling parameters to its destination node. The destination node then uses these propagated scheduling parameters to perform its own local scheduling.

Due to its simplicity and relatively low overhead, this approach has been incorporated into many real-time distributed programming standards. For example, Real-Time CORBA [67] makes extensive use of this approach. However, even within the independent scheduling approach, there are a number of different factors to consider.

One of the most elusive factors (in terms of optimality) is the question of how to derive local scheduling parameters from global scheduling parameters. Specifically, since we are considering real-time systems, the question becomes how to derive local deadlines from global deadlines in order to ensure optimal system performance in terms of deadlines met. There are many different methods for decomposing global end-to-end deadlines in order to derive local deadlines. It is possible to use the end-to-end, or global, deadline to perform local scheduling, however, this approach may underestimate the urgency of components of end-to-end abstractions (since each component is given the urgency of the entire end-to-end abstraction). This may result in excessive delay to said components and thus to deadline misses.

Other approaches include dividing the end-to-end deadline equally among all its component tasks, or dividing the end-to-end deadline in proportion to the execution times of each component. As mentioned before, the "best" method for decomposing end-to-end deadlines is elusive and depends on the application being considered and many other different heuristics. The same set of issues arise if we consider TUFs (see Chapter 1) as our timeliness abstraction. There have been a number of papers addressing this issue [49, 50, 59, 78]. Note that deadline (or TUF) decomposition is essential for independent node scheduling, since it is this technique that allows an end-to-end scheduling problem to be broken down to a series of independent scheduling problems.

Deadline, or TUF, decomposition is not the only issue addressed by researchers in the distributed real-time field. Other important issues include synchronization protocols to ensure precedence constraints are met, scheduling algorithms, and the development of sufficiently tight schedulability analysis for the scheduling and synchronization protocols developed. A good example of previous work that investigates these issues for the independent scheduling paradigm is Sun's thesis [85].

Another work that has addressed the problem of end-to-end scheduling of programming abstractions in distributed real-time systems is [8]. In this paper, the authors consider the flow-shop version of the end-to-end scheduling problem, in which all tasks executed on different processors in the same order. The authors identify two tractable versions of this problem, and suggest a heuristic method for the general NP-hard instance of the problem. The paper, however, does not address fault tolerance or non-synchronous systems. Nor does it consider the behavior of systems during overloads or the possible benefits of collaborative scheduling. All of these factors are taken into consideration in this dissertation.

In [51], Kao *et. al.* present an approach to scheduling soft real-time systems using commercial off the shelf components. However, none of the algorithms used in the paper are specifically designed for scheduling soft real-time systems during overload and hence do not offer reasonable assurances in such scenarios.

Not all the papers on distributed real-time systems have focused on deadline (or TUF) decomposition, and distributed scheduling algorithms. A number of papers have addressed the important issue of developing the necessary schedulability analysis techniques to provide suitable timeliness assurances.

One of the first papers to address the development of scheduling analysis techniques for distributed real-time systems is [88]. In this seminal paper, Tindell *et. al.* develop a method for analyzing distributed real-time systems that many later approaches have built on. The main difficult of schedulability analysis on distributed systems is the fact that the start times of some tasks will depend on the end times of others (since some tasks are invoked when their predecessor task completes and makes a remote invocation).

In order to solve this problem, Tindell *et. al.* proposed the idea of initially setting the start times of a task to the earliest possible completion time of its predecessor task (assuming that each task executes on the processor uninterrupted by other tasks), and then iteratively adjusting these start times as the analysis yields more accurate completion times for tasks in the system. This iterative process is guaranteed to converge if the recurrence relations being iterated over are monotonic in their parameters.

Since Tindell *et. al.*'s seminal paper, several authors have attempted to refine the algorithms and analysis techniques to provide tighter bounds on response times in distributed systems. For example, in [38, 69, 70], the authors propose methods for using jitters and offsets to represent the programming models on a distributed system. The proposed methods allow the authors to develop response time analysis algorithms that can provide tighter bounds on the response times of tasks than previous algorithms. This trend is continued in [72], where the authors attempt to provide an even tighter bound on response times by eliminating the need for jitters. We make use of some of these results while developing our own schedulability analysis techniques for STM based distributed real-time systems in Chapters 7 and 8.

Other previous studies that are relevant to our work include [3–5, 47] which describe schedulability analysis algorithms for stand alone systems programmed using lock-free concurrency control mechanisms. These papers have a relevance to our own STM schedulability analysis algorithms with the major difference being that we concentrate on distributed systems while the papers mentioned address stand alone systems only.

Most of the previous work on distributed real-time systems did not consider either fault tolerance or non-synchronous distributed systems. As previously mentioned, newly emerging distributed systems operate in domains where faults are possible and the system is not always synchronous. One of the proposed directions in this thesis proposal is to determine whether or not it is possible to provide some sort of timeliness assurances in such environments.

Despite the fact the few papers address fault tolerance, there have been some research addressing this issue. For example, the Alpha Kernel was built with fault tolerance in mind. Specifically, the concept of *thread maintenance and repair* (TMAR), was used to monitor the health of end-to-end threads in distributed systems and to recover in case of failure. Chapter 5 in [34] discusses the different algorithms and protocols that can be used to provide TMAR in the Alpha kernel.

More recently, Ravindran *et. al.* [75] develop HUA, an independent node scheduling algorithm for synchronous systems that uses propagated thread scheduling information to perform local scheduling. HUA uses TMAR for fault tolerance and is assumed to execute in a synchronous environment. The problem with independent scheduling is that it achieves its encouragingly low overhead by limiting the information available at each node. Specifically, each node has information about the tasks it hosts only. This lack of information about what is happening on other nodes may result in some decisions that are locally optimal but that compromise global optimality. In this thesis proposal, we study this issue and attempt to qualify the scenarios under which collaborative scheduling is preferable to independent scheduling and whether the significantly higher overhead of collaborative scheduling is justified. This issue is discussed in Chapters 3, 4 and 5.

There have been some recent attempts at designing collaborative scheduling algorithms (e.g., [30, 32, 33, 74], etc). In collaborative scheduling algorithms, scheduling decisions are arrived at after collaboration among the nodes in the distributed system. This collaboration allows better timeliness assurances since nodes can now make informed scheduling decisions to ensure global optimality. However, these algorithm have higher overheads than independent scheduling algorithms and can only benefit systems that can tolerate their higher overhead. In [74], a scheduling algorithm for synchronous systems that uses the collaborative scheduling approach, CUA, is developed. This algorithm uses fast consensus [2] to solve the scheduling problem. The other collaborative scheduling algorithms investigated in this proposal (ACUA [33] , QBUA [30] and DQBUA [32]) are discussed in Chapters 3, 4 and 5 respectively. Another advantage of collaborative scheduling algorithms is that, since global state is shared, at least to some extent, it becomes possible to handle fault tolerance as an integral part of the scheduling algorithm instead of relying on orthogonal thread integrity protocols (such as TMAR). This offers the opportunity to provide better assurances about the behavior of the system in the presences of failures.

From our research in this domain, one fact has become quite obvious. Concurrency control contributes significantly to the complexity of real-time distributed systems. This complexity encompasses both the time complexity of the scheduling algorithms and the semantic complexity of writing correct, deadlock-free code using the standard method of locks and condition variables. We discuss this difficulty and make our case for using an alternative to locks, software transaction memory (or STM), in Chapter 6, however, in Sections 2.0.1 and 2.0.2 we review the literature on this topic.

2.0.1 Alternatives to lock-based programming

Academia, and certain parts of industry, have realized the limitations of lock-based software, thus a number of proposed alternatives to lock-based software exist. The design of lock-free, wait-free or obstruction-free data structures is one such approach. The main problem with this approach is that it is limited to a small set of basic data structures, e.g., [5, 18, 40]. For example, to the best of our knowledge, there is no lock-free implementation of a red-black tree that does not use STM (this does not imply that it is impossible to do so, it is indeed possible, but merely indicates that the difficulty of designing such a complex data structure from basic principles has discouraged researchers from attempting it). Most of the literature on lock-free data structures concentrates on basics such as queues, stacks, and other simple data structures. It should be noted that lock-freedom, wait-freedom and obstruction-freedom are concepts and as such can encompass non lock-based solutions like STM. However, we use these terms in this context to refer to hand crafted code that allows concurrent access to a data structure without suffering from race conditions.

The discrete event model presented in [92, 93] provides an interesting alternative to thread based programming. While interesting and novel, it still remains to be seen whether programmers find the semantics of the model easier than the semantics of thread-based computing. In addition, the requirement of static analysis to determine a partial order on the events makes the system inapplicable to dynamic systems where little or no information is available a priori.

Transactional processing, the semantic ancestor of STM, has been around for a significant period of time and has proven its mettle as a method of providing concurrency control in numerous commercial database products, in addition, it does not place any restriction on the dynamism of the system on which it is deployed. Unfortunately, the use of a distributed commit protocol, such as the two-phase commit protocol, increases the execution time of a transaction and can lead to deadline misses [35]. STM is a lighter-weight version of transactional processing, with no distributed commit protocol required in most cases. As such, it allows us to gain the benefits of transactional processing (i.e., fault tolerance and semantic simplicity), without incurring all its associated overhead.

We believe that STM is an attractive alternative to thread and lock-based distributed programming, since it eliminates many of the conceptual difficulties of lock-based concurrency control at the expense of a justifiable overhead that becomes less significant as the number of processors in the system scales.

2.0.2 Software transactional memory

Since the seminal papers about hardware and software transactional memory were published, renewed interest in the field has resulted in a large body of literature on the topic (e.g, see [9, 55, 63]). This body of work encompasses both purely software transactional memory systems and hybrid systems where software and hardware support for transactional memory are used in conjuncture to improve performance. Despite this large body of work, to the best of our knowledge, only three papers investigate STM for distributed systems [10, 43, 61].

We believe that distributed embedded systems stand to benefit significantly from STM. Such systems are most distinguished by their need to: 1) react to external events asynchronously and concurrently; 2) react to external events in a timely manner (i.e., real-time); and 3) cope with failures (e.g., processors, networks) – one of the raison d'être for building distributed systems. Thus, concurrency that is fundamentally intrinsic to distributed embedded systems naturally motivates the usage of STM. Their need to (concurrently) react timely to external events in the presence of failures is also a compelling reason – such behaviors are very complex to program, reason about, and obtain timing assurances using lock-based concurrency control mechanisms.

There has also been a dearth of work on real-time STM systems. Notable work on transactional memory and lock-free data structures in real-time systems include [3–5, 47, 62]. However, most of these works only consider uni-processor systems (with [47] being a notable exception). In this chapter, we propose to study the issues involved in implementing STM in distributed embedded real-time systems. Past work has shown that STM has lower throughput for systems with a small number of processors compared to fine-grain lock-based solutions but that this difference in performance is quickly reversed as the number of processors scales [39]. This, coupled with easier programming semantics of STM, makes it an attractive concurrency control mechanism for next generation embedded real-time systems with multi-core architectures and high distribution.

With STM, deadlocks are entirely or almost entirely precluded. This will immediately result in significant reductions in the cost of scheduling and resource management algorithms, as distributed dependencies are avoided and no expensive deadlock detection/resolution mechanisms are needed. Implementing higher level programming constructs, like, for example, Hoare's conditional critical regions (or CCR) [46], on top of STM [39], allows programmers to take advantage of the deadlock freedom and simple semantics of STM in their programs.

Chapter 3

Consensus-based collaborative scheduling

3.1 Introduction

In this chapter, we consider the problem of scheduling threads in the presence of the uncertainties mentioned in Chapter 1, focusing particularly on (arbitrary) node failures and message losses. In the model we consider, communication delay and message losses are stochastically described as in [17]. The proposed algorithm is compared to previous distributable thread scheduling algorithms, HUA [75], CUA [74], and ACUA [33].

In this chapter, we target partially synchronous systems, and consider the partially synchronous model in [17], where message delay and message loss are probabilistically described. For such a model, we design a collaborative thread scheduling algorithm called the *Asynchronous Consensus-based Utility accrual scheduling Algorithm* (or ACUA). We show that ACUA satisfies thread time constraints in the presence of crash failures and message losses, is early-deciding (i.e., its decision time is proportional to the actual number of crashes), and has a message and time complexity that compares favorably with other algorithms in its class. Furthermore, we show that ACUA has a better best-effort property — i.e., the affinity for feasibly completing as many high importance threads as possible, irrespective of thread urgency — than past thread scheduling algorithms [74, 75]. We also prove the exception handling properties of ACUA. To the best of our knowledge, this is the first collaborative thread scheduling algorithm designed under a partially synchronous model.

The rest of the chapter is organized as follows: We describe the system models and objectives in Section 3.2. In Section 3.3, we present ACUA. Its analytical properties and an empirical comparison of its performance to other scheduling algorithms are provided in Sections 3.4 and 3.5 respectively. We conclude the chapter in Section 3.6.

3.2 Models and Objective

3.2.1 Models

Distributable Threads. Distributable threads execute in local and remote objects by location-independent invocations and returns. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node.

We assume that execution time estimates of the sections of a thread are known when the thread arrives into the system and are described using TUFs (see our timeliness model). The sequence of remote invocations and returns made by a thread can typically be estimated by analyzing the thread code. The total number of sections of a thread is thus assumed to be known a-priori. The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, \ldots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$.

Timeliness Model. We consider the TUF timeliness model described in Chapter 1.

System Model. We consider a set of nodes $\mathbf{\Pi} = \{1, \cdots, N\}$, with a logical communication channel between each pair of nodes. We assume that each node is equipped with two processors: a processor that executes thread sections on the node and a scheduling co-processor as in [20]. The dual processor assumption is used to reduce ACUA's scheduling overhead. The dual-processor assumption is also reasonable, given the current proliferation of multi-core/CPU chips. We assume that communication links are unreliable, i.e., messages can be lost with probability p, and communication delay is described by some probability distribution.

Bi-directional logical communication channels are assumed to exist between every pair of node. We assume that these basic communication channels may lose messages with probability p, and communication delay is described by some probability distribution.

On top of this basic communication channel, we consider a reliable communication protocol that delivers a message to its destination in probabilistically bounded time provided that the sender and receiver both remain correct, using the standard technique of sequence numbers and retransmissions. We assume that each node is equipped with two processors (a processor that executes thread sections on the node and a scheduling co-processor as in [20]), have access to GPS clocks that provides each node with a UTC time-source with high accuracy (e.g., [23, 36, 84]).

We also assume that each node is equipped with $N-1$ QoS failure detectors (FDs) [17] to monitor the status of all other nodes. On each node, i, these $N-1$ FDs output the nodes they suspect to the list $suspect_i$

Exceptions and Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for thread failures including termination time violations and node failures. When such thread failures occur, the section exception handlers are triggered to restore the system to a safe state. The exception handlers may have time constraints expressed as TUFs.

A handler's TUF's initial time is the time of failure of the handler's thread. The handler's TUF's termination time is relative to its initial time. Thus, a handler's absolute and relative termination times are *not* the same. Each handler also has an execution time estimate. This estimate along with the handler's TUF are described by the handler's thread when the thread arrives at a node. A handler is marked as ready for execution when either its latest start time (see Section 3.3.1) expires, or it receives an explicit invocation from its successor.

Failure Model. The nodes in our system are subject to crash failures. Up to $f_{max} \leq n-1$ nodes can fail in our system. The actual number of failures in the system is denoted as $f \leq f_{max}$.

3.2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property.

3.3 The Proposed Algorithms

3.3.1 ACUA: Algorithm Rationale

ACUA is a collaborative consensus-based scheduling algorithm. Being a collaborative algorithm, ACUA can construct schedules that result in higher system-wide accrued utility by avoiding locally optimal decisions that can compromise system-wide optimality ("local minimums"). It also allows ACUA to respond to node failures by eliminating threads that are affected by the failures, thus allowing the algorithm to gracefully degrade timeliness in the presence of failures.

In ACUA, when a thread arrives into the system, each node suggests a set of threads for rejection from the system based on local scheduling conditions. The nodes must then agree on a set of threads to reject from the system-wide schedule. We formulate this as a consensus problem with the following properties: (a) If a correct node decides on a reject set *rSet*, then some node proposed *rSet*; (b) Nodes do not decide on different reject sets (*Uniform agreement*); (c) Every correct node eventually decides (i.e. termination).

Since ACUA is a consensus-based algorithm, it can only run on systems on which the distributed consensus problem is solvable. In Section 3.3.1, we show that it is possible to design an S class FD, one of the Chandra-Toueg unreliable FDs, on the system model we consider and thus prove that consensus is solvable on that system [14]. Specifically, we show that it is possible to design a FD that provides the semantics of an S class FD *with very high probability* for the *duration of the consensus algorithm.*

Past work [74, 75] had considered the existence of a perfect FD (P class FD), since they considered a synchronous system model. In this work, we use the S class FD (which is weaker than a P class FD) because we consider partially synchronous systems. An S class FD has the following properties: 1) Completeness Property:- There is a time, T_D, after which a failed node is permanently suspected by all nodes; and 2) Accuracy Property:- There is some correct node that is never suspected by all other nodes.

In Section 3.3.1, we describe how end-to-end thread TUFs are decomposed in order to obtain the section TUFs necessary for local scheduling on each node.

Failure Detection

As mentioned in Section 3.2.1, each node is equipped with an aggregate FD consisting of $N-1$ QoS FDs. Assume that this aggregate FD is polled every δ time units in order to learn the state of the system.

From [17] we know that the probability, P_A, that the result of one of the QoS FDs is accurate when it is queried at random is $E(T_G)/E(T_{MR})$, where $E(T_G)$ is the average time that the FD's output remains correct and $E(T_{MR})$ is the average time between consecutive mistakes. We also know that $E(T_G) = E(T_{MR}) - E(T_M)$, where $E(T_M)$ is the average time it takes for the FD to correct an erroneous failure suspicion. Both $E(T_{MR})$ and $E(T_M)$ are input QoS values chosen when designing the FD, thus we can control P_A by choosing appropriate values for these two parameters.

To show that we can implement an S-class FD using the QoS FD in [17], we need to determine when the consensus algorithm needs the service of the FD. The consensus algorithm used in ACUA is the quorum-based algorithm in [66] which requires the service of the FD in line 5 only.

In the worst case, the algorithm takes N rounds (in each of the first $N-1$ rounds an erroneous suspicion of the round coordinator leads to the next round until round N is reached). Let Δ be the communication delay described by the probability density function $delay(t)$ and the cumulative distribution function $DELAY(t)$, the consensus algorithm will spend either Δ to receive the coordinator's estimate or T_D to detect the coordinator's failure.

In the worst case, the consensus algorithm will query the FD n times, where $n = \left\lfloor \frac{N \times T_D}{\delta} \right\rfloor$. We consider each of these queries to be an independent experiment with probability $p = 1 - P_A$ of resulting in an erroneous suspicion. Therefore, the probability that the FD monitoring a single node makes at least one erroneous suspicion during the execution of the algorithm is $P_{FDM} = 1 - bino(0,n,p)$, where $bino(x,n,p)$ is the binomial distribution with parameters n and p. Since there are $N-1$ FDs on each node, the probability that a given node erroneously suspects x nodes is given by $bino(x,N-1,P_{FDM})$ and the probability that a node suspects a majority of the nodes in the system is $\sum_{i=\frac{N-1}{2}+1}^{N-1} bino(i,N-1,P_{FDM})$. Using this analysis, we constructed a FD that suspected a majority of nodes with probability 1.5×10^{-110} for realistic settings. We believe this probability is too low to be of practical concern for the time scales we consider.

Since it is not practically possible for a node to erroneously suspect a majority of other nodes during the execution of the consensus algorithm, the set of nodes not suspected by all nodes in the system have to intersect in at least one node. That node is never suspected by any of the other nodes in the system, thus satisfying the accuracy property of an S class FD. In addition, the T_D detection time of our FD satisfies the completeness property of an S class FD. Therefore, we are able to implement an S class FD with very high probability on our system during the execution of our consensus algorithm.

Numerical Example

We now present a numerical example to show that the analysis in Section 3.3.1 can result in an S class FD. As in [17], we assume that the message delay is modeled by an exponential distribution with mean and variance of 0.02 seconds. We also assume that the probability of message loss is 0.01 and that there are 100 nodes in the system. In designing the FD, we chose $T_D = 1$ second, $E(T_M) = 0.5$ seconds and $E(T_{MR}) = 1$ month. We also assume that the consensus algorithm queries the FD every 10ms when it needs it. Therefore we can conclude that the consensus algorithm will make $n = 1 * 100/(10 \times 10^{-3}) = 10000$ queries to the FD.

$E(T_G) = E(T_{MR}) - E(T_G) = 2591999.5$. Therefore, $P_A \simeq 0.99999981$, and $p = 1.92901234541409 \times 10^{-7}$. As described in Section 3.3.1, the probability that the FD detector makes at least one mistake is $P_{FDM} = 1 - bino(0,n,p)$, $P_{FDM} = 0.000964041279072125$. Using this value of P_{FDM}, we compute the probability that a node suspects a majority of other nodes in the system, $\sum_{i=\frac{N-1}{2}+1}^{N-1} bino(i,N-1,P_{FDM})$, to be $1.50224277975635 \times 10^{-110}$. We believe that this probability is too low to be of practical concern for the time scales we are considering. To get a perspective, note that the number of protons in the whole universe is a 24 digit number. Therefore, since it is not practically possible for a node to erroneously suspect a majority of other nodes during the execution of the consensus algorithm, the set of nodes not suspected by all nodes in the system have to intersect in at least one node. That node is never suspected by any of the other nodes in the system, thus satisfying the accuracy property of an S class FD.

TUF Decomposition

Thread time constraints are expressed using TUFs. The termination time of each section belonging to a thread needs to be derived from that thread's end-to-end termination time. This derivation should ensure that if all the section termination times are met, then the end-to-end termination time of the thread will also be met.

For the last section of a thread, we derive its termination time as the thread's termination time. The termination time of the other sections is the latest start time of the section's successor minus the communication delay. Thus the section termination times of a thread T_i, with k sections, is:

$$S^i_j.tt = \begin{cases} T_i.tt & j = k \\ S^i_{j+1}.tt - S^i_{j+1}.ex - D & 1 \le j \le k-1 \end{cases}$$

where $S^i_j.tt$ denotes section S^i_j's termination time, $T_i.tt$ denotes T_i's termination time, and $S^i_j.ex$ denotes the estimated execution time of section S^i_j. The communication delay, which we denote by D above, is a random variable Δ, as mentioned in Section 3.3.1. Therefore, the value of D can only be determined probabilistically. This implies that if each section meets the termination times computed above, the whole thread will meet its termination time with a certain, high, probability. This is further explored in Section 3.5.

As mentioned in Section 3.2.1, each handler has a TUF that specifies its **relative** termination time, $S^h_j.X$. However, a handler's **absolute** termination time is relative to the time it is released, more specifically, the **absolute** termination time of a handler is equal to the sum of the **relative** termination time of the handler and the failure time t_f (which cannot be known a priori). In order to overcome this problem, we delay the execution of the handler as much as possible which allows us to delay the execution of the exception handlers as much as possible, thus leaving room for more important threads. Therefore, in the equations below we replace t_f with $S^i_k.tt$, the termination time of thread i's last section:

$$S^h_j.tt = \begin{cases} S^i_k.tt + S^h_j.X + T_D + t_a & j = k \\ S^h_{j+1}.tt + S^h_j.X + D & 1 \le j \le k-1 \end{cases}$$

where $S^h_j.tt$ denotes section handler S^h_j's termination time, $S^h_j.X$ denotes the relative termination time of section handler S^h_j, t_a is a correction factor corresponding to the execution time of the scheduling algorithm, and T_D is the time needed to detect a failure by our QoS FD. From this decomposition, we compute start times for each handler:

$$S^h_j.st = \left\{ \; S^h_j.tt - S^h_j.ex \quad 1 \le j \le k \right.$$

where $S^h_j.ex$ denotes the estimated execution time of section handler S^h_j. Thus, we assure the feasible execution of the exception handlers of failed sections, in order to revert the system to a safe state.

Algorithm Description

Algorithm 1 shows the general structure of ACUA. Algorithm 1 is triggered when a thread arrives into the system or when a node fails. When ACUA is triggered, each node constructs a local schedule (line 5). In lines 6-14 each node suggests a set of threads for rejection based on the local schedule it constructs in line 5. In line 15, the nodes send the set of threads they suggest for rejection to all other nodes in the system. Each node then waits for a certain time period to collect the suggestions that other nodes send (lines 15-16). Using these suggestions, each node makes a decision about which set of threads should be rejected from the system (line 18). A consensus protocol is then started in order to reach agreement among the nodes about the set of threads that will be rejected, using the decision each node made in line 18 as input to the consensus protocol (line 19). After reaching agreement, the nodes remove the set of rejected threads from their waiting queue (line 20) and construct a new local schedule containing the remaining threads (line 21).

An important part of ACUA is how it selects a set of threads for rejection locally (lines 7-14). ACUA distinguishes between threads that become unschedulable due to local overloads, and threads that become unschedulable in order to accommodate a newly arrived thread. This is necessary because a newly arrived thread can only be accepted into the system if *all* its future head nodes accept its sections. Thus, if some nodes reject other threads' sections in order to accommodate the arriving thread, and other nodes reject the sections of the arriving thread, the new thread should not be accepted into the system and the sections rejected to accommodate the new thread's sections on some nodes should be allowed to execute normally.

Algorithm 1: ACUA: ACUA on each node i

```
1:  input: $\sigma_r^i$; // $\sigma_r^i$: unordered ready queue;
2:  input: $\sigma_p$; // $\sigma_p$ : previous schedule;
3:  output $\sigma_i$; // $\sigma_i$: schedule;
4:  Initialization: $\Sigma_i = \emptyset$; $w_i = \emptyset$;
5:  $\sigma_i$ = ConstructSchedule($\sigma_r^i$);
6:  if i is head node for newly arrived thread j then
7:      $\sigma_{tmp}$ = ConstructSchedule($\sigma_r^i - S_j^i$);
8:      if $S_j^i \notin \sigma_i$ then
9:          $rSet = 0 \cup (\sigma_p - \sigma_i)$;
10:     else
11:         $tmp = (\sigma_p - \sigma_{tmp})$;
12:         $rSet = 1 \cup (\sigma_p - (\sigma_i - S_j^i) - tmp) \cup \perp \cup tmp$;
13: else
14:     $rSet = \emptyset \cup (\sigma_p - \sigma_i)$;
15: send($rSet_i, i, t$) to all;
16: upon receive($rSet_j, j$) until $2D$ do
17:     $\Sigma_i = \Sigma_i \cup rSet_j$;
18: $w_i$=DetRejectSet($\Sigma_i$);
19: $w_i$=UniformConsensus($w_i$);
20: UpdateSectionSet($w_i, \sigma_r^i$);
21: $\sigma_i$ =ConstructSchedule($\sigma_r^i$);
22: $\sigma_p = \sigma_i$;
23: return $\sigma_i$;
```

Lines 7-12 perform this function. If the section of the newly arrived thread is not part of the constructed schedule, it cannot be responsible for the elimination of other threads from the system. Thus the difference between the current schedule and the previous schedule is the set of threads that the node proposes for rejection (lines 8-9). On the other hand, if a section of the newly arrived thread is part of the schedule, we need to differentiate between two possible causes for rejecting threads: 1) overload conditions may render some threads unschedulable and 2) the newly arrived thread may render some threads unschedulable.

The former set can be determined by constructing a schedule without considering S^i_j (line 7) and then subtracting that set from the set of previously schedulable threads (line 11). On line 12 we place a separator, $\perp$, between the set of threads rendered unschedulable due to overload and the set of threads rendered unschedulable due to the acceptance of a section of the newly arrived thread. Note that nodes indicate whether they accept, reject, or are not responsible for the sections of a newly arrived thread by prepending 1, 0 and $\emptyset$ to their suggestions respectively.

Using this additional information, the problem mentioned above can be eliminated by only eliminating threads rendered unschedulable by an arriving thread if ***all*** its future head nodes accept the thread. The details of this functionality is contained in the function *DetRejectSet*. Note that the timeout value on line 16 is a stochastic value, thus even if none of the nodes fail, there is a non-zero probability that some nodes do not receive the suggestions of all other nodes. This is further addressed in Section 3.5.

Algorithm 2: ACUA: DetRejectSet on node *i*

```
1:  input: $\Sigma_i$; // $\Sigma_i$: set of suggestions for rejection.
2:  output $w_i$; // $w_i$: rejection set output.
3:  accept=true;
4:  $w_i = \emptyset$;
5:  for each future head node, j, of newly arrived thread do
6:      $tmp_j$=retrieve node $j$'s entry from $\Sigma_i$;
7:      if $head(tmp_j)=0$ then
8:          accept=false;
9:  for each node j do
10:     $rSet_j$=retrieve node $j$'s entry from $\Sigma_i$;
11:     $rSet_j = rSet_j$ - first element in $rSet_j$;
12:     if j is a future head node then
13:         if accept=true then
14:             $w_i=w_i \cup$ elements before and after $\perp$ in $rSet_j$;
15:         else
16:             $w_i=w_i \cup$ only elements after $\perp$ in $rSet_j$;
17:     else
18:         $w_i=w_i \cup rSet_j$;
19:     if node j is a head node for thread set $\Gamma$ with a section on node i then
20:         if node i does not receive node j's suggestion then
21:             $w_i=w_i \cup \Gamma$;
22: return $w_i$;
```

Algorithm 2 describes how nodes determine the set of threads to suggest for rejection from the system. The algorithm first checks whether the newly arrived thread has been accepted into the system by all future head nodes (lines 5-8). Lines 10 and 11 retrieve the suggestion of node j and remove the first element. Lines 12-16 determine which threads to consider for rejection based on the fact that threads rendered unschedulable by the newly arrived thread on some nodes should only be rejected if ***all*** head nodes accept the sections of the newly arrived thread. Line 18 adds the set of threads that non-head nodes suggest for rejection. Finally, lines 19-21 suggests threads for rejection if they have a section hosted on the current node and the current node does not receive any suggestions from one of the previous, current, or future head nodes of the threads. This is done because a node suspects those nodes it does not receive suggestions from to have failed, and thus suggests for elimination the threads that are hosted by them. The uniform consensus algorithm we use is described in [66].

We now turn our attention to the scheduling algorithm that nodes use to construct a local schedule. This algorithm is encapsulated by the function *ConstructSchedule* (see Algorithm 3). The algorithm takes a list of sections, and constructs a total order with each section's global Potential Utility Density (or PUD). The global PUD of a section is the ratio of the utility of the thread that the section belongs to, to the sum of the remaining execution times of all the thread's sections. The algorithm examines each section in PUD-order, including them in the schedule, and testing for schedule feasibility. If infeasible, the inserted section is rejected, and the process is repeated until all sections are examined. Note that we construct a total order on *global* Potential Utility Density (PUD) in order to attempt to maximize system-wide accrued utility. This can be seen in line 8 of the algorithm, where the execution time of the whole thread, $T_i.ex$, is used instead of the execution time for each individual section, $S_i.ex$, when computing PUD. The algorithm for *UpdateSectionSet* involves a simple removal of the rejected threads from a node's ready queue.

3.4 Experimental Results

We performed a series of simulation experiments on ns-2 [64] to compare the performance of ACUA to CUA and HUA in terms of Accrued Utility Ratio (AUR) and Termination-time Meet Ratio (TMR). We define AUR as the ratio of the accrued utility (the sum of U_i for all completed threads) to the utility available (the sum of U_i for all available jobs) and TMR as the ratio of the number of threads that meet their termination time to the total number of threads in the system. We considered threads with three segments. Each thread starts at its origin node with its first segment. The second segment is a result of a remote invocation to some node in the system, and the third segment occurs when the thread returns to its origin node to complete its execution. The periods of these threads are fixed, and we vary their execution times to obtain a range of utilization ranging from 0 to 200%. In order to make the comparison fair, all the algorithms were simulated using a synchronous system model, where communication delay varied according to an exponential distribution with mean and standard deviation 0.02 seconds but could not exceed an upper bound of 0.5 seconds. Our

Algorithm 3: ACUA: ConstructSchedule

```
1: input: $\sigma_r$,$\sigma_p$,$H$; output $\sigma$;
2: Initialization: $t = t_{cur}$;$\sigma = \emptyset$;$HandlerIsMissed = $ fasle;
3: for each $S_i \in \sigma_p$ such that $S_i \notin \sigma_r$ do
4:     Insert($S_j^h$,$H$,$S_j^h.tt$);
5: $\sigma = H$;
6: for each $S_i \in \sigma_r$ do
7:     if $S_{j-1}^i.tt + D + S_j^i.ex \leq S_j^i.tt$ then
8:         $S_i.PUD = min\left(\frac{U_i(t+T_i.ex)}{T_i.ex}, \frac{U_i^h(t+T_i.ex+T_i^h.ex)}{T_i.ex+T_i^h.ex}\right)$
9:     else
10:        $S_i.PUD = 0$
11: $\sigma_{tmp}$=sortByPUD($\sigma_r$);
12: for each $S_i \in \sigma_{tmp}$ from head to tail do
13:     if $S_i.PUD \geq 0$ then
14:         Insert($S_i$,$\sigma$,$S_i.tt$);
15:         Insert($S_i^h$,$\sigma$,$S_i^h.tt$);
16:         if $Feasible(\sigma) = false$ then
17:             Remove($S_i$,$\sigma$,$S_i.tt$);
18:             if $S_i^h \notin H$ then
19:                 Remove($S_i^h$,$\sigma$,$S_i^h.tt$);
20:     else
21:         break;
22: $\sigma_p = \sigma$;
23: return $\sigma$;
```

system consisted of fifty client nodes and five servers. In all the experiments we perform, the utilization of the system is considered the *maximum* utilization experienced by any node. While conducting our experiments, we considered three different thread sets.

Thread set I. In the first thread set we consider, our thread set parameters — i.e., section execution times, thread termination times, and thread utility — are chosen to highlight the better distributed best-effort properties of ACUA. The strength of ACUA lies in its ability to give priority to threads that will result in the most system-wide accrued utility. Therefore, the thread set that highlights this property is one that contains threads that would be given low priority on a node if local scheduling is performed but should be assigned high priority due to the system-wide utility that they accrue to the system. Therefore, our first thread set contains high utility threads that have one section with above average execution time (resulting in low PUD for that section) and other sections with below average execution times (resulting in high PUD for those section). Such thread sets test the ability of the algorithm to take advantage of collaboration to avoid making locally optimal decisions that would compromise global optimality.

Thread set II. In the second thread set, the section execution times, thread utilities and termination times are generated using uniform random variables. This thread set represents the random load that a system may experience during normal execution. Using this thread set, we compare the performance of the algorithms to see how they respond to “normal” thread sets.

Thread set III. Finally, since the second threat set is neutral to all algorithms in the sense that it is randomly generated and does not fit the performance of any particular algorithm better than the other, and the first thread set favors ACUA since it specifically contains thread sets that are best scheduled by collaborative algorithms to avoid "local minimums", our last thread set is designed to contain threads that can be scheduled to accrue maximum utility without requiring collaboration. Our intention in choosing such a thread set is to see how ACUA compares to other algorithms when collaboration is not required.

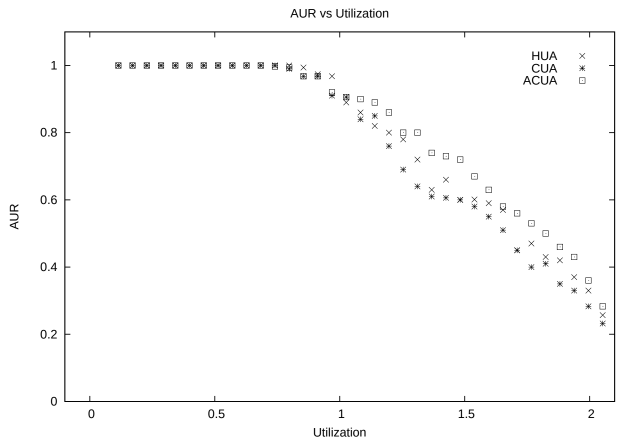

Figure 3.1: ACUA: AUR vs. Utilization (no failures)

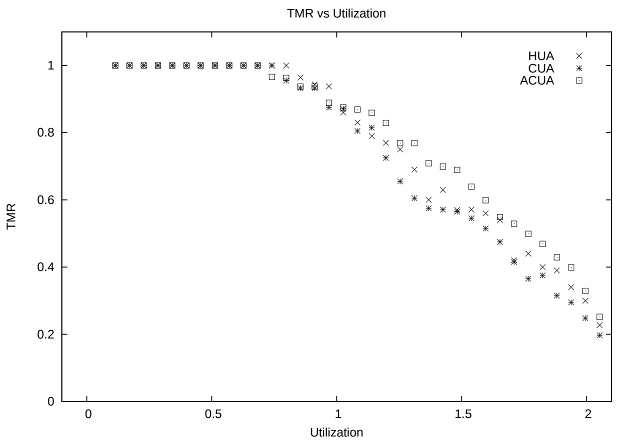

Figure 3.2: ACUA: TMR vs. Utilization (no failures)

Figures 3.1 and 3.2 show the result of our AUR and DSR experiments in the absence of node failure for thread set I. As Figures 3.1 and 3.2 show, the performance of ACUA during underloads is similar to that of other distributed real-time scheduling algorithms. However, during overloads, ACUA begins to outperform other algorithms due to its better best effort property. During overloads, ACUA accrues, on average, 10% more utility that CUA, and 6% more utility than HUA. The maximum difference between the performance of ACUA and the other algorithms in our experiment was the 16% difference between ACUA's and CUA's AUR at the 1.31 system load point. Throughout our experiment, the performance of HUA was the closest to ACUA with the difference in performance between these two algorithms getting more pronounced as system load increases (the largest difference in performance is 12% and occurs at about 1.48 system load). Another interesting aspect of the experiment is that CUA and ACUA do not accrue 100% utility during all cases of underload. As the load on the system approaches 1.0 some deadlines are missed because their overhead becomes more significant at this point. This is true to a lesser extent for non-collaborative scheduling algorithms such as HUA due to their lower overhead.

Figures 3.3 and 3.4 show the effect of failures on ACUA when thread set I is used. In these experiments we programmatically fail $f_{max} = 0.2N$ nodes — i.e., we fail 20% of the client nodes. From Figure 3.3, we see that failures do not degrade the performance of ACUA

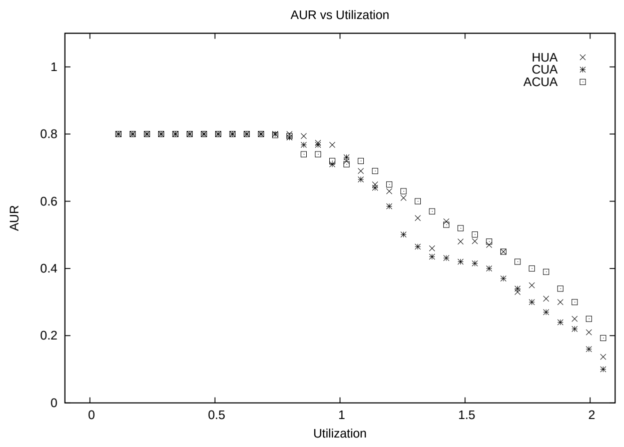

Figure 3.3: ACUA: AUR vs. Utilization (failures)

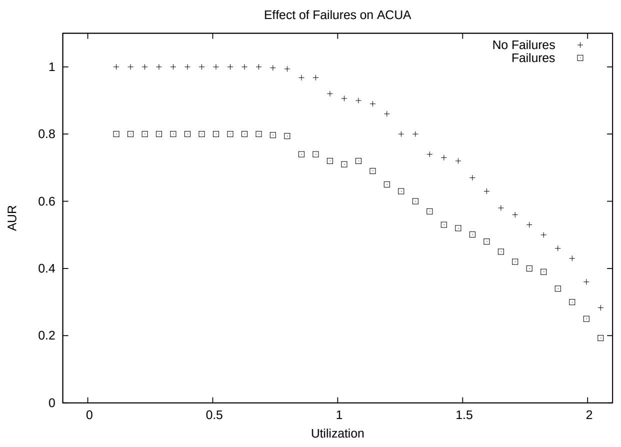

Figure 3.4: ACUA: Effect of failures on ACUA

compared to other scheduling algorithms — i.e., the relationship between the utility accrued by ACUA to the utility accrued by other scheduling algorithms remains relatively the same in the presence of failures.

In Figure 3.4 we compare the behavior of ACUA in the presence of failure to its behavior in the absence of failure. As can be seen, ACUA's performance suffers a degradation in the presence of failures. As can be seen from the figure, the difference in performance of ACUA in the presence of failure is most pronounced during underloads, and becomes less pronounced as the system load is increased. The reason for this is that during underloads all threads are feasible and therefore the failure of a particular node deprives the system of the utility of all the threads that have a section hosted on that node. However, during overloads, not all sections hosted by a node are feasible, thus the failure of that node only deprives the system of the utility of the feasible threads that have a section hosted by that node. This amount is less than would occur if the system were deprived of the utility of all threads hosted on the node and leads to a less pronounced effect on the performance of ACUA in the presence of failures.

In Figures 3.5, 3.6, 3.7 and 3.8, the same experiments as above are repeated using thread set II. As can be seen, the results follow a similar pattern to the experiments for thread set I, but now the difference between ACUA and other algorithms is not so pronounced. This occurs because the randomly generated thread set is likely to contain some threads that need collaborative scheduling but not as many as in the thread set that we specifically designed to contain such threads.

In Figures 3.9, 3.10, 3.11 and 3.12, the same experiments as above are repeated using thread set III. In these experiments, the best performing algorithm is HUA since it has the least overhead (it does not perform any collaborative scheduling). The performance of the other algorithms are similar to each other.

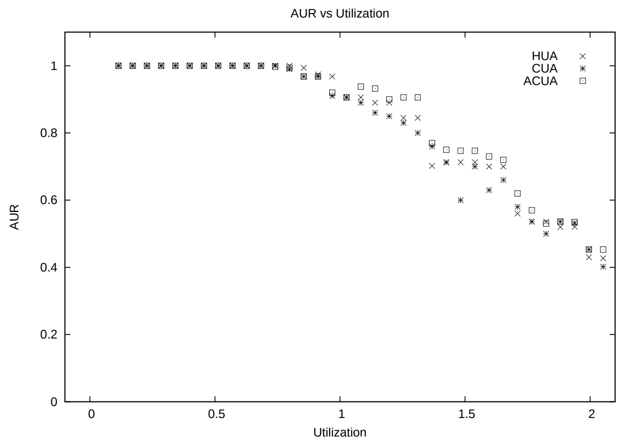

Figure 3.5: ACUA: AUR vs. Utilization (no failures), thread set II

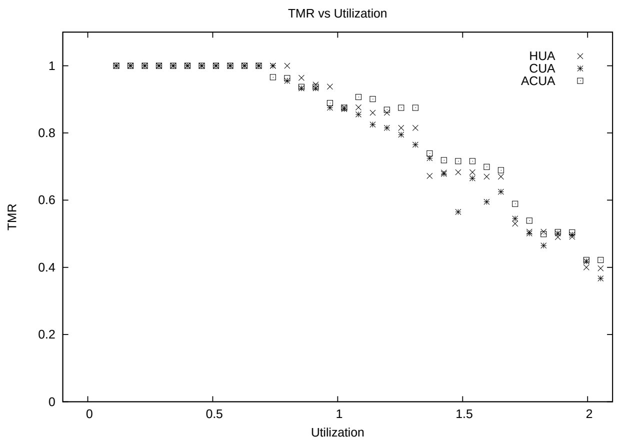

Figure 3.6: ACUA: TMR vs. Utilization (no failures), thread set II

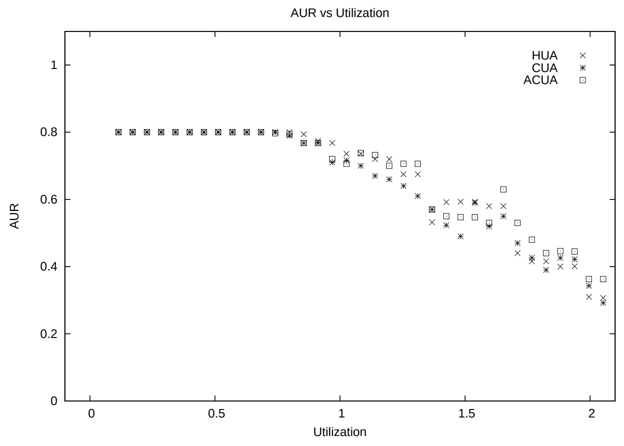

Figure 3.7: ACUA: AUR vs. Utilization (failures), thread set II

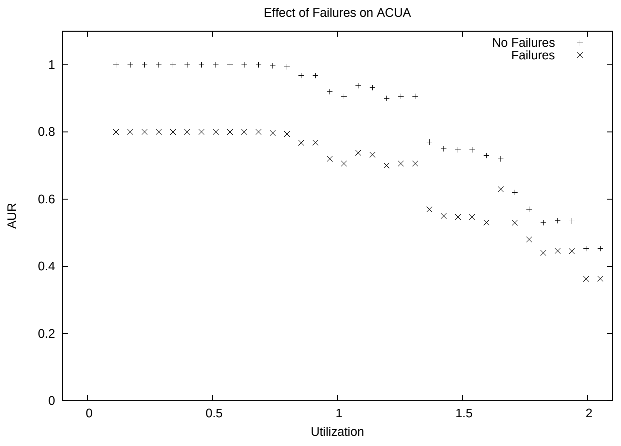

Figure 3.8: ACUA: Effect of failures of ACUA, thread set II

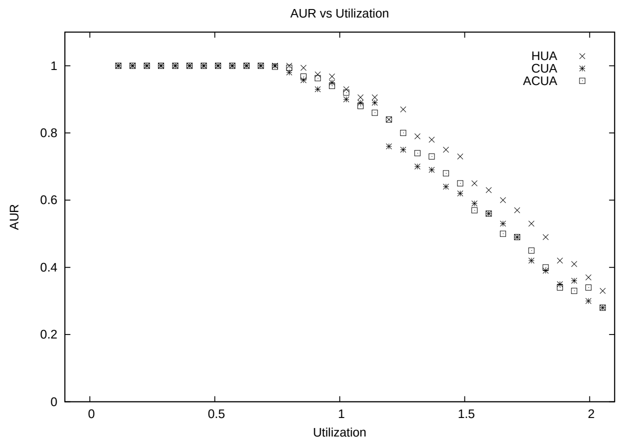

Figure 3.9: ACUA: AUR vs. Utilization (no failures), thread set III

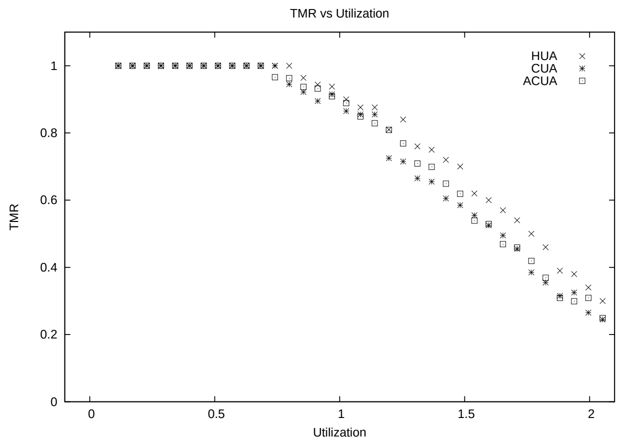

Figure 3.10: ACUA: TMR vs. Utilization (no failures), thread set III

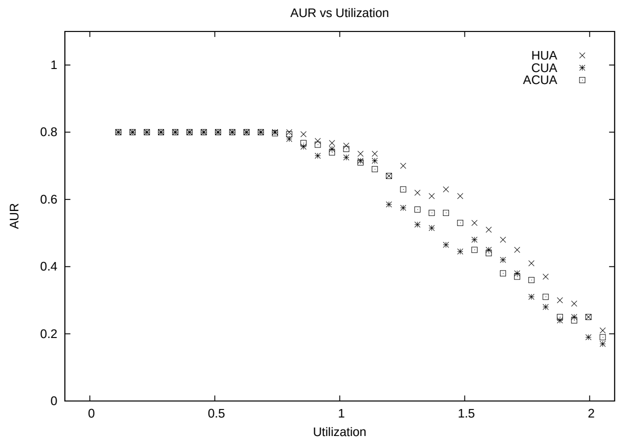

Figure 3.11: ACUA: AUR vs. Utilization (failures), thread set III

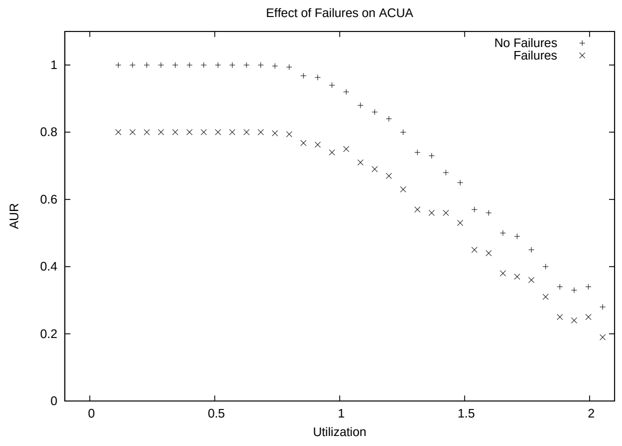

Figure 3.12: ACUA: Effect of failures of ACUA, thread set III

We now turn our attention to studying the effect of ACUA's overhead on scheduling. In order to do so, we fix the communication delay at 60ms, fix section execution times (as shown on the x-axis of Figure 3.13), and vary the period of the threads to obtain different utilizations. Theoretically, ACUA should meet all deadlines in underloaded systems (i.e., systems with a load less than or equal to one). However, in practise, the overhead of the algorithm interferes with the running tasks and so causes deadline misses before full system utilization is reached. In this experiment, we attempt to determine the system utilization at which the first deadline misses occur (we call this the Deadline Miss Load, or DML). This experiment gives us a notion on the thread execution time scales that can withstand the overhead of ACUA.

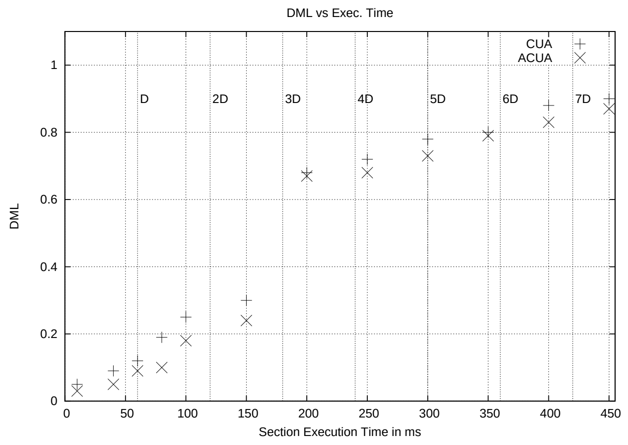

Figure 3.13: ACUA: DML vs. Section Execution Time

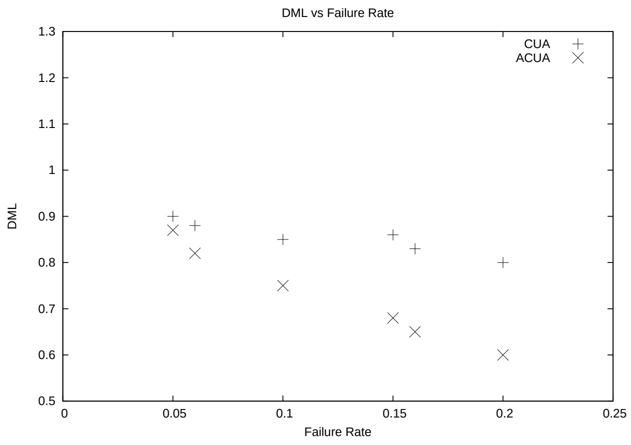

Figure 3.14: ACUA: DML vs. Section Execution Time

As can be seen in Figure 3.13, the system performs poorly for section execution times below $3D$, where D is the communication delay. This occurs because that is the minimum time period required for the algorithm to work properly. It should also be noted that CUA performs slightly better than ACUA, this occurs because CUA depends on the existence of a fast failure detector and thus has lower overhead. In both algorithms, as the section execution times increase, the DML begins to approach one.

We performed another experiment to determine the effect of failures on the section execution time scales that can tolerate the overhead of the algorithms. The result is depicted in Figure 3.14. In this experiment, we fixed the section execution time at $7D$, fixed the failure rate, i.e., the percentage of nodes that fail, (shown on the x-axis of Figure 3.14), and varied the period to determine the utilization, DML, at which the first deadline is missed due to time overruns rather than node failures. Note that this new definition of DML is necessary because we wish to determine the effect of failures on the overhead, thus we need to exclude the deadline misses that occur because nodes hosting certain sections fail.

As can be seen in Figure 3.14, the DML for both algorithms drop, however, the DML for ACUA drops faster than that of CUA. The reason for this is that the time complexity of

ACUA is $O(f\Delta + nk)$ (see Section 3.5), while that of CUA is $O(D + df + nk)$ [74]. Where f is the number of failures, D and Δ are the communications delays for CUA and ACUA respectively (note that Δ is a random variable, while D is not), n is the number of nodes, k is the number of sections in each thread and d ($d \ll D$) is the detection time of the fast failure detector used in CUA. As can be noted, both time complexities are a function of the number of failures f, however, for CUA f is multiplied by d while for ACUA it is multiplied by Δ, so for CUA each additional failure introduces an additional d ($d \ll D$) overhead, while for ACUA, the system suffers a full communication delay, Δ, for each failure. Therefore, CUA scales better in the presence of failure.

However, a number of factors need to be taken into account when understanding this result. First, for sufficiently large section execution times, the DML for both algorithms approach one. Also, for cases that require collaboration (such as thread set I used in the first set of experiments described in this chapter), ACUA will outperform CUA despite its higher overhead because it makes better system-wide decisions. Finally, since CUA is designed for a fully synchronous system the algorithm breaks down if any of the synchrony assumptions are violated, in contrast to ACUA which has higher coverage since it is designed for partially synchronous systems.

3.5 ACUA properties

We compare the best-effort properties of ACUA, CUA [74], and HUA [75]. In HUA (an *independent* node scheduling algorithm), thread sections are scheduled locally at each node they arrive at using their propagated scheduling parameters. The local scheduler is a modified version of DASA [21], which uses the heuristic of favoring tasks with a high utility to execution time ratio, i.e., high PUD, when constructing the schedule. These modifications allow HUA to manage the scheduling of exception handlers in case of thread failure.

In CUA (a *collaborative* scheduling algorithm), when a thread arrives, its sections are sent to all its future head nodes. Each node constructs its schedule locally according to a modified version of DASA. The nodes then cooperate with each other to reach agreement on a system-wide set of threads eligible for execution. Basically, this agreement step involves the elimination of any threads that have any of their sections missing from the global schedule.

We quantify the best-effort property by introducing the concept of DASA Best Effort (or DBE) property:

Definition 1. *Consider a distributed scheduling algorithm $\mathcal{A}$. DBE is defined as the property that $\mathcal{A}$ orders its threads in non-increasing order of global PUD while considering them for scheduling and schedules all feasible threads in the system in that order.*

Note that the ***DBE*** property is essential for any UA algorithm that attempts to maximize system-wide accrued utility by favoring tasks that offer the most utility for the least amount of execution time (which is the heuristic used by DASA [21]).

Lemma 1. *HUA, does not have the **DBE** property.*

Proof. The proof is by counterexample. Assume that a system has two nodes, n_1 and n_2, and two threads, T_1 and T_2. Assume that each thread has two sections, one hosted on each of the nodes. Let the sections be S_1^1 and S_1^2 for T_1 and S_2^1 and S_2^2 for T_2. Assume that both threads have end-to-end step-down TUFs, with the utility for T_1 being 5 and the utility of T_2 being 6. Also assume that both threads arrive at n_1 at t_0. Assume that the execution times of S_1^1, S_1^2, S_2^1 and S_2^2 are 2, 3, 3 and 1 time units respectively and that both threads have a relative termination time of 5.

The parameters above ensure that only one of the threads can be scheduled successfully. Therefore, an algorithm that has the ***DBE*** property would choose T_2 for execution since its global PUD, $\frac{6}{4} = 1.5$, is greater than the PUD of T_1, $\frac{5}{5} = 1$. Note that the ***DBE*** property will result in a system-wide accrued utility of 6 in this case.

In contrast, HUA computes the PUD of the *sections* of each thread hosted on each node when constructing its schedule [75]. Since the PUD of S_1^1, $\frac{5}{2} = 2.5$, is greater than the PUD of S_2^1, $\frac{6}{3} = 2$, the scheduler on n_1 will choose S_1^1 for scheduling first. By the time S_1^1 has finished execution, $t_0 + 3$, releasing S_2^1 for execution will mean that it will finish past the global termination time of T_2 ($T_0 + 5$). Thus, only T_1 will execute with a resulting accrued utility of 5 for the system.

Thus HUA does not have the ***DBE*** property. □

Lemma 2. *CUA does not have the DBE property.*

Proof. CUA does not have the ***DBE*** property became it does not schedule all feasible threads in the system. For example, if two nodes host sections of two threads, T_1 and T_2, during overloads, one node may schedule the section belonging to T_2 at the expense of that belonging to T_1 and the other may schedule the section belonging to T_1 at the expense of that belonging to T_2. Since CUA excludes threads from the system if they are missing any of their sections and both of the above threads have one of their sections missing, both threads will be excluded from the system. This is unnecessary since excluding one thread will render the other schedulable, thus the algorithm does not schedule all feasible threads and therefore does not have the ***DBE*** property. □

Theorem 3. *ACUA has the **DBE** property for threads that can be delayed $O(f\Delta + nk)$ (see Lemma 5) and are still schedulable.*

Proof. ACUA overcomes the issue mentioned in Theorem 1 because it uses the PUD of the entire thread when constructing local schedules on each node. Thus sections that are excluded are those with the least *system-wide* PUD. In other words, the threads in ACUA are considered in non-increasing order of global PUD for scheduling. In addition, ACUA overcomes the issue mentioned in Theorem 2 by preventing an arriving thread from eliminating other threads if at least one of the nodes that will be hosting a future head of the arriving thread does not accept that section for scheduling. The details of this procedure are explained in Algorithms 1 and 2. This allows ACUA to schedule all feasible threads. Thus all feasible threads that can tolerant the scheduling overhead of ACUA and still remain feasible will be scheduled in non-increasing order of global PUD. The theorem follows from Definition 1. □

Theorem 4. *ACUA can tolerate up to $f_{max} = n-1$ faulty processors.*

Proof. This follows directly from the fault tolerant property of the S class based consensus algorithm in [66] which we use in our work. □

Lemma 5. *ACUA has time complexity $O(f\Delta + nk)$.*

Proof. Lines 5 and 7 in Algorithm 1 have complexity $O(k^2)$ where k is the maximum number of sections in the ready queue of system nodes. Lines 8-15 have constant complexity, lines 16-17 have complexity 2Δ, line 18 has complexity $O(nk)$, line 19 has complexity $O((f+1)\Delta)$, line 20 has complexity $O(k)$ and line 21 has complexity $O(k^2)$. Therefore, the algorithm has actual complexity of $3k^2 + 2\Delta + nk + (f+1)\Delta + k$, which is asymptotically $O(f\Delta + nk)$ if we consider k a constant. □

This time complexity compares favorably with the time complexity of CUA, which is $O(D + df + nk)$ [74], asymptotically. However, the value of the time complexity of CUA is lower than that of ACUA since it makes the additional assumption of the existence of a fast FD [2]. In addition, Δ is a random variable, thus the timing guarantee for ACUA is stochastic in nature.

Lemma 6. *ACUA has message complexity $O(fn^2)$.*

Proof. Lines 16-17 in Algorithm 1 have message complexity n (one for each suggested rejection set sent by a node). Line 19 has message complexity $n^2(f+1)$ since each round has a message cost of n^2. The algorithm is early deciding so it will take $f+1$ rounds [66]. Therefore the actual message cost of the algorithm is $n + n^2(f+1)$, which is asymptotically $O(fn^2)$. □

Lemma 7. *The message size in ACUA is smaller than that in CUA for well behaved systems.*

Proof. The input to the consensus algorithm in ACUA is the set of rejected threads while the input to the consensus algorithm in CUA is the set of schedulable threads. Since the set of rejected threads should be smaller than the set of accepted threads in well behaved systems, we claim that the message size in ACUA is smaller than that in CUA. □

Lemma 8. *If each section of a thread meets its derived termination time (see Section 3.3.1), then under ACUA, the entire thread meets its termination time with high, computable probability, p_{suc}.*

Proof. Since the termination times derived for sections are a function of communication delay, and this communication delay is a random variable with CDF $DELAY(t)$, the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation in Section 3.3.1 are violated during runtime.

Let D be the communication delay used in the derivation of section termination times. The probability that D is violated at runtime is $p = 1 - DELAY(D)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above D is $p_{suc} = bino(0, k, p)$. Thus, the probability that the thread meets its termination time is also $p_{suc} = bino(0, k, p)$. □

Lemma 9. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by ACUA with high, computable, probability p_{norej}.*

Proof. Since the nodes are all underloaded and no nodes fail, Algorithm 3 ensures that all sections will be accepted. Thus, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, D, (see Algorithm 1 in Section 3.3.1). This can occur due to one of two reasons; 1) the broadcast message (line 15), that indicates the start of the consensus algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to other nodes in the system during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(D)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast start of consensus message reaches all nodes is $P_{tmp} = bino(0, N, p)$ where $bino(x, n, p)$ is the binomial distribution with parameters n and p. If this message is received, a node waits for messages from all other nodes. The probability that none of these messages arrive after the timeout is $tmp = bino(0, N, p)$. Since there are N nodes, the probability that none of these nodes miss a message is $bino(N, N, tmp)$. Therefore the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes receive suggestions from all other nodes in response to this start of consensus message i.e. $p_{norej} = bino(N, N, tmp) \times P_{tmp}$. □

Theorem 10. *If all nodes are underloaded, no nodes fail (i.e. $f = 0$), and threads can be delayed $O(f\Delta + nk)$ time units once and still be schedulable, ACUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg}.*

Proof. By Lemma 9, no threads will be considered for rejection from a fault free, underloaded system with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 3. Thus, by Lemma 8, each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with X threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

ACUA takes $O(f\Delta + nk)$ time units to determine a newly arrived thread's schedulability. If this delay causes any of the thread's sections to miss their termination times, the thread will not be schedulable. We require that a thread suffer this delay *once* because we assume that there is a scheduling co-processor on each node. Thus, the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 11. *ACUA is an early deciding algorithm that achieves consensus on the system-wide execution eligible thread set in a partially synchronous system with virtually certain probability.*

Proof. Since the consensus algorithm in [66], on which we base our algorithm, is early deciding so is our algorithm. In addition, we show in Section 3.3.1 that we can provide an S class FD with very high probability during the execution of our algorithm (with probability of error 1.50×10^{-110}), therefore the S class FD based consensus algorithm in [66] executes on our system with virtually certain probability. Since the input to the consensus algorithm is the set of threads to reject from the system, at its completion all nodes will agree on the set of threads to reject from their schedules and hence on the system-wide set of execution eligible threads. □

Theorem 12. *If $n - f$ nodes do not crash, are underloaded, and all incoming threads can be delayed $O(f\Delta + nk)$ and still be schedulable, ACUA meets the execution time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. By Theorem 11, ACUA achieves system-wide consensus on the set of schedulable threads. By Lemma 9, the probability that none of the threads hosted by the surviving nodes are rejected is, $p_{norej} = bino(N-f, N-f, tmp) \times tmp$ where $tmp = bino(0, N-f, p)$ and $p = 1 - DELAY(D)$. Thus all sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 8, this implies that each of these threads, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p^j_{suc}$. Thus, the probability that all the remaining threads are accepted is $P_{alg} = P_{tmp} \times p_{norej}$. □

Definition 2 (Section Failure). *A section, S^i_j, is said to have failed when one or more of the previous head nodes of S^i_j's thread (other than S^i_j's node) has crashed.*

Lemma 13. *If a node hosting a section, S^i_j, of thread T_i fails at time t_f, every correct node will include handlers for thread T_i in H by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for ACUA calculated per the analysis in Theorem 5.*

Proof. Since the QoS FD we use detects a failed node in T_D time units [17], all nodes detect the failure of the failed node at time $t_f + T_D$. As a result, ACUA is triggered and excludes T_i from the system because nodes will not receive any suggestions from node j (see lines 19-21 of Algorithm 2). Consequently, Algorithm 3 will include the section handlers for this thread in H (see lines 3-4 of Algorithm 3). Execution of ACUA completes in time t_a and thus all handlers will be included in H by time $t_f + T_D + t_a$. □

Again it should be noted that t_a is a stochastic value and therefore the timeliness property above is probabilistic in nature. The probability that a particular value of t_a can be met is easy to obtain by considering the CDF of Δ when conducting the analysis in Theorem 5.

Lemma 14. *If a section S_i, where $i \neq k$, fails at time t_f (per Definition 2) and section S_{i+1} is correct, then under ACUA, its handler S^h_i will be released no earlier than S^h_{i+1}'s completion and no later than $S^h_{i+1}.tt + D + S^h_i.X - S^h_i.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires; or 2) an explicit invocation is made by the handler's successor.

For the first case, we know from the analysis in Section 3.3.1 that the start time of S^h_i is $S^h_{i+1}.tt + S^h_j.X + D - S^h_j.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S^h_j.X \geq S^h_j.ex$ (otherwise the handler would not be schedulable), $S^h_{i+1}.tt + S^h_j.X + D - S^h_j.ex > S^h_{i+1}.tt$, and this satisfies the lower bound of the theorem.

For the second case, an explicit message has arrived indicating the completion of S^h_{i+1}. Since the message was sent, this means that $S^h_{i+1}.tt$ has already passed, thus satisfying the theorem lower bound. Further, the message should have arrived D time units after S^h_{i+1} finishes execution (i.e., at $S^h_{i+1}.tt + D$), since $S^h_{i+1}.tt + D \leq S^h_{i+1}.tt + D + S^h_i.X - S^h_i.ex$ (as $S^h_i.X \geq S^h_i.ex$), thus satisfying the upper bound. □

Lemma 15. *If a section S_i fails (per Definition 2), then under ACUA, its handler S^h_i will complete no later than $S^h_i.tt$ (barring S^h_i's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system-wide schedulable set previously constructed. This means that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S^h_i.tt$, respectively (lines 13-19, Algorithm 3). When some previous head node of S_i's thread fails, ACUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 3 will include S^h_i in H and construct a feasible schedule containing S^h_i (lines 3-21). Since the schedule is feasible and S^h_i is inserted to meet $S^h_i.tt$ (line 4), then S^h_i will complete by time $S^h_i.tt$ □

Theorem 16. *When a thread fails, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kD + T_D + t_a$.*

Proof. The LIFO property follows from Lemma 14. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 14), thus we guarantee LIFO execution of the handlers. Lemma 15 shows that all correct handlers complete in bounded time. Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 13) and ACUA guarantees that all these sections will complete before their termination times (Lemma 15). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 3.3.1) is $T_i.X + \sum_i S_i^h.X + kD + T_D + t_a$. The theorem follows. □

3.6 Conclusion

We presented a best-effort utility accrual scheduling algorithm, ACUA, for scheduling distributable real-time threads in partially synchronous systems. We compared ACUA in terms of its best-effort property, and message and time complexity to two previous thread scheduling algorithms including CUA and HUA. We showed that ACUA has a better best-effort property during overloads than HUA and CUA, and has message and time complexities that are comparable to CUA (which is in its class). We also showed the exception handling properties of ACUA.

Both CUA and HUA have optimal best-effort properties during underloads. The two algorithms, as well as ACUA, achieve optimal total utility during underloads (for CUA and ACUA this property comes with the caveat that the threads should be able to tolerate the algorithm overheads). However, during overloads, Lemmas 1, 2 and Theorem 3 show that ACUA has better best-effort semantics. In addition, if we choose to implement ACUA on a totally synchronous system, it will have exactly the same message and time complexity as CUA (since we can now use the fast consensus algorithm in [2] instead of the quorum-based algorithm in [66]) and yet possess better best-effort properties than CUA during overloads. In addition, all ACUA's stochastic properties will become deterministic.

Chapter 4

Quorum-based collaborative scheduling

4.1 Introduction

In this chapter, we consider the problem of scheduling threads in the presence of the uncertainties mentioned in Chapter 1, focusing particularly on (arbitrary) node failures and message losses. In the model we consider, communication delay and message losses are stochastically described as in [17]. The proposed algorithm is compared to previous distributable thread scheduling algorithms, HUA [75], CUA [74], and ACUA [33].

We present a collaborative scheduling algorithm called the *Quorum-Based Utility Accrual scheduling* (or QBUA) algorithm. The algorithm considers the partially synchronous model in [17], and uses a Quorum set of nodes for majority agreement on constructing system-wide thread schedules. We show that QBUA satisfies thread time constraints in the presence of node crash failures and message losses, has efficient message and time complexities that compare favorably with other algorithms in its class, and superior timeliness than past algorithms including CUA and HUA. We also show that the algorithm's lower overhead, in the presence of failure, enables it to allow more threads to benefit from its superior timeliness, than that allowed by past algorithms.

The rest of the chapter is organized as follows: We describe the system models and objectives in Section 4.2. In Section 4.3, we present QBUA. Its analytical properties and an empirical comparison of its performance to other scheduling algorithms are provided in Sections 4.4 and 4.5 respectively. We conclude the chapter in Section 4.6.

4.2 Models and Objective

4.2.1 Models

Distributable Threads. As in Chapter 3, we use the distributable thread abstraction as our programming model.

Timeliness Model. We employ the TUF timeliness model described in Chapter 1.

System Model. We consider a networked embedded system to consist of a set of client nodes $\Pi^c = \{1, 2, \cdots, N\}$ and a set of server nodes $\Pi = \{1, 2, \cdots, n\}$ (*server* and *client* are logical designations given to nodes to describe the algorithm's behavior). Bi-directional logical communication channels are assumed to exist between every client-server and client-client pair. We also assume that these basic communication channels may lose messages with probability p, and communication delay is described by some probability distribution.

On top of this basic communication channel, we consider a reliable communication protocol that delivers a message to its destination in probabilistically bounded time provided that the sender and receiver both remain correct, using the standard technique of sequence numbers and retransmissions. We assume that each node is equipped with two processors (a processor that executes thread sections on the node and a scheduling co-processor as in [20]), and have access to GPS clocks that provides each node with a UTC time-source with high accuracy (e.g., [23, 36, 84]).

We also assume that each node is equipped with $N-1$ QoS failure detectors (FDs) [17] to monitor the status of all other nodes. On each node, i, these $N-1$ FDs output the nodes they suspect to the list $suspect_i$

Exceptions and Abort Model. We employ the same exception and abort model described in Chapter 3.

Failure Model. Nodes are subject to crash failures. When a process crashes, it loses its state memory — i.e., there is no persistent storage. If a crashed client node recovers at a later time, we consider it a new node since it has already lost all of its former execution context. A client node is *correct* if it does not crash; it is *faulty* if it is not correct. In the case of a server crash, it may either recover or be replaced by a new server assuming the same server name (using DNS or DHT — e.g, [26] — technology). We model both cases as server recovery.

Since crashes are associated with memory loss, recovered servers start from their initial state. A server is *correct* if it does not fail; it is *faulty* if it is not correct. DQBUA tolerates up to $N-1$ client failures and up to $f^s_{max} \leq n/3$ server failures. The actual number of failures is denoted as $f^s \leq f^s_{max}$ for servers and $f \leq f_{max}$ where $f_{max} \leq N-1$ for clients.

4.2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property.

4.3 Proposed Algorithm

4.3.1 QBUA: Algorithm Rationale

QBUA is a collaborative scheduling algorithm. Thus, QBUA can construct schedules that result in higher system-wide accrued utility by avoiding locally optimal decisions that can compromise system-wide optimality ("local minimums"). It also allows QBUA to respond to node failures by eliminating threads that are affected by the failures, thus allowing the algorithm to gracefully degrade timeliness in the presence of failures. There are two types of scheduling events that are handled by QBUA; a) local scheduling events and b) distributed scheduling events.

Local scheduling events are handled locally on a node without consulting other nodes. Examples of local scheduling events are section completion and section handler expiry events. For a full list of local scheduling events, please see Algorithm 10. Distributed scheduling events need the participation of all nodes in the system to handle them. In this work, only two distributed scheduling events exit; a) the arrival of a new thread into the system and b) failure of a node.

A node that detects a distributed scheduling event sends a START message to all other nodes requesting their scheduling information so that it can compute a System Wide Executable Thread Set (or SWETS). Nodes that receive this message, send their scheduling information to the requesting node and wait for schedule updates (which are sent to them when the requesting node computes a new system-wide schedule). This may lead to contention if several different nodes detect the same distributed scheduling event concurrently.

For example, when a node fails, many nodes may detect the failure concurrently. It is superfluous for all these nodes to start an instance of QBUA. In addition, events that occur in quick succession may trigger several instances of QBUA when only one instance can handle all of those events. To prevent this, we use a quorum system to arbitrate among the nodes wishing to run QBUA. In order to perform this arbitration, the quorum system examines the time-stamp of incoming events. If an instance of QBUA was granted permission to run *later* than an incoming event, there is no need to run another instance of QBUA since information about the incoming event will be available to the version of QBUA already running (i.e., the event will be handled by that instance of QBUA).

In order to perform this functionality, QBUA requires timeliness information for each of the sections it schedules. As mentioned in Section 4.2.1, end-to-end thread timeliness requirements are described using TUFs.

4.3.2 Algorithm Description

We use the same method of TUF decomposition as mentioned in Chapter 3. As mentioned above, whenever a distributed scheduling event occurs, a node attempts to acquire permission from the quorum system to run a version of QBUA. After the quorum system has arbitrated among the nodes contending to execute QBUA, the node that acquires the "lock" executes Algorithm 4. In Algorithm 4, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units[1] for all nodes in the system to respond by sending their local scheduling information (line 2). After collecting this information, the node computes SWETS (line 3) using Algorithm 8. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule as a result of the scheduling event).

Algorithm 4: QBUA: Compute SWETS

```
1: Broadcast start of algorithm message, START;
2: Wait $2T$ collecting replies from other nodes;
3: Construct SWETS using information collected;
4: Multicast change of schedule to affected nodes;
5: return;
```

The algorithm that client nodes run when attempting to acquire a "lock" on running a version of QBUA, Algorithm 5, is loosely based on Chen's solution for FTME [16]. Upon the arrival of a distributed scheduling event, a node tries to acquire a "lock" on running QBUA (the try_1 part of the algorithm that starts on line 3). The first thing that the node does (lines 4-5) is check if it is currently running an instance of QBUA that is in its information collection phase (line 2 in Algorithm 4). If so, the new event that has occurred can simply be added to the information being collected by this version of QBUA. However, if no current instance of QBUA is being hosted by the node, or if the instance of QBUA being hosted has passed its information collection phase, then the event may have to spawn a new instance of QBUA (this starts at line 6 in the algorithm).

The first thing that Algorithm 5 does in this case is send a time-stamped request to the set of server nodes, Π, in the system (lines 8-10). The time-stamp is used to inform the quorum nodes of the time at which the event was detected by the current node. Beginning at line 3, Algorithm 5 collects replies from the servers. Once a sufficient number of replies have arrived (line 14), Algorithm 5 checks whether its request has been accepted by a sufficient ($\lceil \frac{2n}{3} \rceil$ see Section 4.5) number of server nodes. If so, the node computes SWETS (lines 15-16).

[1] T is communication delay derived from the random variable describing the communication delay in the system.

On the other hand, if an insufficient number of server nodes support the request, two possibilities exist. The first possibility is that another node has been granted permission to run an instance of QBUA to handle this event. In this case, the current node does not need to perform any additional action and so releases the “lock” it has acquired on some servers (lines 17-21).

The second possibility is that the result of the contention to run QBUA at the servers was inconclusive due to differences in communication delay. For example, assume that we have 5 servers and three clients wishing to run QBUA and all three clients send their request to the servers at the same time, also assume different communication delay between each server and client. Due to these communication differences, the messages of the clients may arrive in such a pattern so that two servers support client 1, another 2 servers support client 2 and the last server supports client 3. This means that no client’s request is supported by a sufficient — i.e., $\frac{2n}{3}$ — number of server nodes. In this case, the client node sends a YIELD message to servers that support it and an INQUIRE message to nodes that do not support it (line 22-28) and waits for more responses from the server nodes to resolve this conflict. Lines 30-35 release the “lock” on servers after the client node has computed SWETS, lines 36-38 are used to handle the periodic cleanup messages sent by the servers and lines 39-41 respond to the START of algorithm message (line 1, Algorithm 4).

Algorithm 6 is run by the servers, the function of this algorithm is to arbitrate among the nodes contending to run QBUA so as to minimize the number of concurrent executions of the algorithm. Since there may be more than one instance of QBUA running at any given time, the server nodes keep track of these instances using three arrays. The first array, $c_{owner}[]$, keeps track of which nodes are running instances of QBUA, the second, $t_{owner}[]$, stores the time at which a node in $c_{owner}[]$ sends a request to the servers (i.e., the time at which that node detects a certain scheduling event), and $t_{grant}[]$ keeps track of the time at which server nodes grant permission to client nodes to execute QBUA. Also, a waiting queue for each running instance of QBUA is kept in $R_{wait}[]$.

When a server receives a message from a client node, it first checks to see if this is a stale message (which may happen due to out of order delivery). A message from a client node, c_1, that has a time-stamp older than the last message received from c_1 has been delivered out of order and is ignored (line 7-8). Starting at line 9, the algorithm begins to examine the message it has received. If it is a REQUEST message, the server checks if the time-stamp of the event triggering the message is *less* than the time at which a client node was *granted* permission to run an instance of QBUA. If such an instance exists, a new instance of QBUA is not needed since the event will be handled by that previous instance of QBUA. Algorithm 6, inserts the incoming request into a waiting queue associated with that instance of QBUA and sends a message to the client (lines 10-13).

However, if no current instance of QBUA can handle the event, a client’s request to start an instance of QBUA is granted (lines 14-18). If a client node sends a YIELD message, the server revokes the grant it issued to that client and selects another client from the waiting queue

Algorithm 5: QBUA: QBUA on client node i

```
1: timestamp; // time stamp variable initially set to nil
2: upon thread arrival or detection of a node failure:
3:   try_1:
4:     if a current version of QBUA is waiting for information from other nodes then
5:         Include information about event when computing SWETS;
6:     else
7:         timestamp ← GetTimeStamp;
8:         for all $r_j \in \Pi$ do
9:             resp[j] ← (nil,nil);
10:            send (REQUEST, timestamp) to $r_j$;
11:        repeat
12:            wait until [received (RESPONSE, owner, t) from some $r_j$];
13:            if ($c_1 \neq$ owner or timestamp = t) then resp[j] ← (owner, t);
14:            if among resp[], at least m of them are not (nil, nil) then
15:                if at least m elements in resp[] are ($c_1$, t) then
16:                    return Compute SWETS;
17:                else if at least m elements in resp[] agree about a certain node then
18:                    for all $r_k \in \Pi$ such that resp[k] $\neq$ (nil, nil) do
19:                        if resp[k].owner = $c_1$ then
20:                            send (RELEASE,timestamp) to $r_k$;
21:                    Skip rest of algorithm; //Event is already being handled
22:                else
23:                    for all $r_k \in \Pi$ such that resp[k] $\neq$ (nil, nil) do
24:                        if resp[k].owner = $c_1$ then
25:                            send (YIELD,timestamp) to $r_k$;
26:                        else
27:                            send (INQUIRE,timestamp) to $r_k$;
28:                        resp[k] ← (nil,nil);
29:        until forever ;
30:      exit_1:
31:        oldtimestamp ← timestamp;
32:        timestamp ← GetTimeStamp;
33:        for all $r_k \in \Pi$ do
34:            send (RELEASE, oldtimestamp) to $r_j$;
35:            return;
36: upon receive (CHECK, t) from $r_j$
37:   if for all instances of QBUA running on this node, timestamp $\neq$ t then
38:       send (RELEASE, t) to $r_j$;
39: upon receive (START) from some client node
40:   Update $RE_j^i$ for all sections;
41:   send $\sigma_j$ and $RE_j^i$'s to requesting node;
```

for that event (lines 21-31). This part of the algorithm can only be triggered if the result of the first round of contention to run QBUA is inconclusive (as discussed when describing Algorithm 5). Recall that this inconclusive contention is caused by different communication delays that allow different requests to arrive at different severs in different orders. However, all client requests for a particular instance of QBUA are queued in $R_{wait}[]$, therefore, when a client sends a YIELD message, servers are able to choose the highest priority request (which we define as the request with the earliest time-stamp and use node id as a tie breaker). Thus, we guarantee that this contention will be resolved in the second round of the algorithm. Lines 32-34 show servers' response to INQUIRE messages and lines 35-39 show the clean up procedures to remove stale messages. As can be seen on line 36, when a node that is currently running an instance of QBUA fails, HandleFailure($c_{owner}[i]$,$c_{owner}[]$,$t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$) is called to handle this failure. Algorithm 7 shows the details of this function. If the waiting queue corresponding to this instance of QBUA, $R_{wait}[i]$, is empty, then there are no other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$ and so the system is cleared of this instance of QBUA (lines 1-3). Otherwise, there are other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$, or another concurrent event, and therefore the failure of $c_{owner}[i]$ results in selecting another node from the waiting queue $R_{wait}[i]$ to run QBUA to handle this event (lines 4-9).

Algorithm 8 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 2 in Algorithm 4). It performs two basic functions, first, it computes a system wide order on threads by computing their global Potential Utility Density (PUD). It then attempts to insert the remaining sections of each thread, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After the insertion of each thread, the schedule is checked for feasibility. If it is not feasible, the thread is removed from SWETS (after scheduling the appropriate exception handler if necessary).

First we need to define the global PUD of a thread. Assume that a thread, T_i, has k sections denoted $\{S_1^i, S_2^i, \cdots, S_k^i\}$. We define the global remaining execution time, GE_i, of the thread to be the sum of the remaining execution times of each of the thread's sections. Let $\{RE_1^i, RE_2^i, \cdots, RE_k^i\}$ be the set of remaining execution times of T_i's sections, then $GE_i = \sum_{j=1}^{k} RE_j^i$. Assuming that we are using step-down TUFs, and T_i's TUF is $U_i(t)$, then its global PUD can be computed as $T_i.PUD = U_i(t_{curr} + GE_i)/GE_i$, where U is the utility of the thread and t_{curr} is the current time. Using global PUD, we can establish a system wide order on the threads in non-increasing order of "return on investment". Thus allowing us to consider them for scheduling in an order that attempts to maximize accrued utility [21].

In Algorithm 8, each node, j, sends the node running QBUA its current local schedule σ_j^p. Using these schedules, the node can determine the set of threads, Γ, that are currently in the system. Both these variables are inputs to the scheduling algorithm (lines 1 and 2 in Algorithm 8). In lines 3-6, the algorithm computes the global PUD of each thread in Γ.

Before we schedule the threads, we need to ensure that the exception handlers of any thread

Algorithm 6: QBUA: QBUA on server node i

```
1: $c_{owner}[]$; Array of nodes holding lock to run QBUA
2: $t_{owner}[]$; $t_{owner}[i]$ contains time-stamp of event that triggered QBUA for node in $c_{owner}[i]$
3: $t_{grant}[]$; $t_{grant}[i]$ contains time at which node in $c_{owner}[i]$ was granted lock to run QBUA
4: $R_{wait}[]$; $R_{wait}[i]$ is waiting queue for instance of QBUA being run by $c_{owner}[i]$;
5: upon receive ($tag$, $t$)
6:   $CurrentTime \leftarrow$ GetTimeStamp;
7:   if ($c_1$, $t'$) appears in ($c_{owner}[]$,$t_{owner}[]$) or $R_{wait}[]$ then
8:       if $t < t'$ then Skip rest of algo; //This is an old message
9:   if $tag$ = $REQUEST$ then
10:      if $\exists\ t_{grant} \in t_{grant}[]$ such that $t \leq t_{grant}$ then
11:          send (RESPONSE, $c$, $t_{grant}$) to $c_1$; //where $c \leftarrow c_{owner}[i]$, such that $t_{grant}[i] = t_{grant}$;
12:          Enqueue ($c_1$, $t$) in $R_{wait}[i]$, such that $t_{grant}[i] = t_{grant}$;
13:          Skip rest of algorithm;
14:      else
15:          AddElement($c_{owner}[]$,$c_1$);
16:          AddElement($t_{owner}[]$,$t$);
17:          AddElement($t_{grant}[]$,$CurrentTime$);
18:          send (RESPONSE, $c_1$, $t$) to $c_1$;
19:  else if $tag$ = $RELEASE$ then
20:      Delete entry corresponding to $c_1$, $t$ from $c_{owner}[]$, $t_{owner}[]$, $t_{grant}[]$, and $R_{wait}[]$;
21:  else if $tag$ = $YIELD$ then
22:      if ($c_1$, $t$) $\in$ ($c_{owner}[]$, $t_{owner}[]$) then
23:          For $i$, such that ($c_1$, $t$) = ($c_{owner}[i]$, $t_{owner}[i]$)
24:          Enqueue ($c_1$, $t$) in $R_{wait}[i]$;
25:          ($c_{wait}$,$t_{wait}$) $\leftarrow$ top of $R_{wait}[i]$;
26:          $c_{owner}[i] \leftarrow c_{wait}$; $t_{owner}[i] \leftarrow t_{wait}$;
27:          $t_{grant}[i] \leftarrow CurrentTime$;
28:          send (RESPONSE, $c_{wait}$, $t_{wait}$) to $c_{wait}$;
29:      if $c_1 \notin c_{owner}[]$ then
30:          ($c$,$t_p$) $\leftarrow$ ($c_{owner}[i]$,$t_{owner}[i]$), for min $i$ such that $t \leq t_{grant}[i]$;
31:          send (RESPONSE, $c$, $t_p$) to $c_1$;
32:  else if $tag$ = $INQUIRE$ then
33:      ($c$,$t_p$) $\leftarrow$ ($c_{owner}[i]$,$t_{owner}[i]$), for min $i$ such that $t \leq t_{grant}[i]$;
34:      send (RESPONSE, $c$, $t_p$) to $c_1$;
35: upon suspect that $c_{owner}[i]$ has failed:
36:   HandleFailure($c_{owner}[i]$,$c_{owner}[]$,$t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$);
37: periodically:
38:   $\forall\ c_{owner} \in c_{owner}[]$:
39:     send (CHECK, $t_{owner}$) to $c_{owner}$; //NB. $t_{owner}$ is the entry in $t_{owner}[]$ that corresponds to $c_{owner}$.
```

Algorithm 7: QBUA: HandleFailure($c_{owner}[i]$,$c_{owner}[]$,$t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$)

```
1: if $R_{wait}[i]$ is empty then
2:     remove $c_{owner}[i]$'s entry from $c_{owner}[]$, $t_{owner}[]$, $t_{grant}[]$;
3:     Delete $R_{wait}[i]$;
4: else
5:     $CurrentTime \leftarrow$ GetTimeStamp;
6:     ($c_{wait}$,$t_{wait}$) $\leftarrow$ top of $R_{wait}[i]$;
7:     $c_{owner}[i] \leftarrow c_{wait}$; $t_{owner}[i] \leftarrow t_{wait}$;
8:     $t_{grant}[i] \leftarrow CurrentTime$;
9:     send (RESPONSE, $c_{wait}$, $t_{wait}$) to $c_{wait}$;
```

Algorithm 8: QBUA: ConstructSchedule

```
1:  input: $\Gamma$; //Set of threads in the system
2:  input: $\sigma_j^p$, $H_j \leftarrow$ nil; //$\sigma_j^p$: Previous schedule of node $j$, $H_j$: set of handlers scheduled
3:  for each $T_i \in \Gamma$ do
4:      if for some section $S_j^i$ belonging to $T_i$, $t_{curr} + S_j^i.ex > S_j^i.tt$ then
5:          $T_i.PUD \leftarrow 0$;
6:      else $T_i.PUD \leftarrow \frac{U_i(t_{curr}+GE_i)}{GE_i}$;
7:  for each task $el \in \sigma_j^p$ do
8:      if el is an exception handler for section $S_j^i$ then Insert($el$, $H_j$, $el.tt$);
9:  $\sigma_j \leftarrow H_j$;
10: $\sigma_{temp} \leftarrow$ sortByPUD($\Gamma$);
11: for each $T_i \in \sigma_{temp}$ do
12:     $T_i.stop \leftarrow false$;
13:     if did not receive $\sigma_j$ from node hosting one of $T_i$'s sections $S_j^i$ then
14:         $T_i.stop \leftarrow true$;
15:     for each remaining section, $S_j^i$, belonging to $T_i$ do
16:         if $T_i.PUD > 0$ and $T_i.stop \neq true$ then
17:             Insert($S_j^i$, $\sigma_j$, $S_j^i.tt$);
18:             if $S_j^h \notin \sigma_j^p$ then Insert($S_j^h$, $\sigma_j$, $S_j^h.tt$);
19:             if isFeasible($\sigma_j$)=false then
20:                 $T_i.stop \leftarrow true$;
21:                 Remove($S_k^i$, $\sigma_k$, $S_k^i.tt$) for $1 \leq k \leq j$;
22:                 if $S_j^i \notin \sigma_j^p$ then Remove($S_j^h$, $\sigma_j$, $S_j^h.tt$);
23: for each $j \in N$ do
24:     if $\sigma_j \neq \sigma_j^p$ then Mark node $j$ as being affected by current scheduling event;
```

that has already been accepted into the system can execute to completion before its termination time. We do this by inserting the handlers of sections that were part of each node's previous schedule into that node's current schedule (lines 7-9). Since these handlers were part of σ_j^p, and QBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will execute to completion before their termination times.

In line 10, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 11-21). In lines 13-14 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If the thread can contribute non-zero utility to the system and the thread has not been rejected from the system, we insert its sections into the scheduling queue of their corresponding node (line 17).

After inserting the section into its corresponding ready queue (at a position reflecting its termination time), we check to see whether this section's handler had been included in the previous schedule of the node. If so, we do not insert the handler into the schedule since this has been already taken care of by lines 7-8. Otherwise, the handler is inserted into its corresponding ready queue (line 18). Once the section, and its handler, have been inserted into the ready queue, we check the feasibility of the schedule (line 19). If the schedule is infeasible, we remove the thread's sections from the schedule (line 21). However, we first check to see whether the section's handler was part of a previous schedule before we remove it (line 22). We perform this check before removing the handler because if the handler was part of a previous schedule, then its section has failed and we should keep its exception handler for clean up purposes. Finally, if the schedule of any node has changed, these nodes are marked to have been affected by the current instance of QBUA (lines 23-24). It is to these nodes that the current node needs to multicast the changes that have occurred (line 4, Algorithm 4). In order to test the feasibility of a schedule, we need to check if all the sections in the schedule can complete before their derived termination times.

We use a function *isFeasible* in Algorithm 8 to determine the feasibility of the schedules we construct. Algorithm 9 contains the details of this algorithm. In order to determine whether a schedule is feasible or not, we need to check if all the sections in the schedule can complete before their derived termination times.

The loop on lines 3-6 examines each section in the schedule and attempts to determine whether or not it can complete before its termination time. Since QBUA is executed before sections have actually arrived at some of the nodes in the system, we need to provide an estimate for the starting time of each section on a node. A section can start immediately when it arrives at a node or it may wait some time if other sections are scheduled for execution before it. Therefore, the start time of a section is the maximum of the termination time of the last section scheduled to execute before it ($CumExeTime$) and the time it arrives at the node, which we estimate using the termination time of its predecessor section, plus the communication delay $S_{j-1}^i.tt+T$. To this start time, we add the execution time of the section, $S_j^i.ex$, to obtain an estimate of the expected completion time of the section (line 4).

We then check if this value is greater than the derived termination times of the section; if so, the schedule is infeasible (lines 6-7).

Algorithm 9: QBUA: *isFeasible*

```
1: input: $\sigma$;output: true or false;
2: Initialization: CumExeTime=$t_{curr}$;
3: for each section $S^i_j \in \sigma$ do
4:     $CumExeTime \leftarrow \max(CumExeTime, S^i_{j-1}.tt + T) + S^i_j.ex$; // If j=1, then $T = 0$;
5:     if $CumExeTime > S^i_j.tt$ then return false;
6: return true;
```

QBUA's dispatcher is shown in Algorithm 10. Only two scheduling events result in collaborative scheduling, viz: the arrival of a thread into the system, and the failure of a node, all other scheduling events are handled locally. Since we are talking about a partially synchronous system, the FD we use to detect node failures can make mistakes. Thus, QBUA may be started due to an erroneous detection of failure. The this can be reduced by designing a QoS FD [17] with appropriate QoS parameters.

Algorithm 10: QBUA: Event Dispatcher on each node *i*

```
1  Data: schedevent, current schedule $\sigma_p$;
2  switch schedevent do
3      case invocation arrives for $S^i_j$
4          mark segment $S^i_j$ ready;
5      case segment $S^i_j$ completes
6          remove $S^i_j$ from $\sigma_r$,$\sigma_p$;
7          remove $S^h_j$ from $H$;
8          set $RE^i_j$ to zero;
9      case $S^h_j \in H$ and $S^h_i.st$ expires
10         mark handler $S^h_j$ ready;
11     case downstream handler $S^h_{j+1}$ completes
12         mark handler $S^h_j$ ready;
13     case handler $S^h_j$ completes
14         remove $S^h_j$ from $\sigma_p$,$H$;
15         notify scheduler for $S^h_{j-1}$;
16     case new thread, $T_i$, arrives
17         if origin node, send segments $S^i_j$ to all;
18         pass event to QBUA;
19     case node failure detected
20         pass event to QBUA;
21 execute first ready segment in $\sigma_p$;
```

4.4 Experimental Results

We performed a series of simulation experiments on ns-2 [64] to compare the performance of QBUA to ACUA, CUA and HUA in terms of Accrued Utility Ratio (AUR) and Termination-time <u>M</u>eet <u>R</u>atio (TMR). We define AUR as the ratio of the accrued utility (the sum of U_i for all completed threads) to the utility available (the sum of U_i for all available jobs) and TMR as the ratio of the number of threads that meet their termination time to the total number of threads in the system. We considered threads with three segments. Each thread starts at its origin node with its first segment. The second segment is a result of a remote invocation to some node in the system, and the third segment occurs when the thread returns to its origin node to complete its execution. The periods of these threads are fixed, and we vary their execution times to obtain a range of utilization ranging from 0 to 200%. In order to make the comparison fair, all the algorithms were simulated using a synchronous system model, where communication delay varied according to an exponential distribution with mean and standard deviation 0.02 seconds but could not exceed an upper bound of 0.5 seconds. Our system consisted of fifty client nodes and five servers. In all the experiments we perform, the utilization of the system is considered the *maximum* utilization experienced by any node. While conducting our experiments, we considered the same three different thread sets considered in Chapter 3.

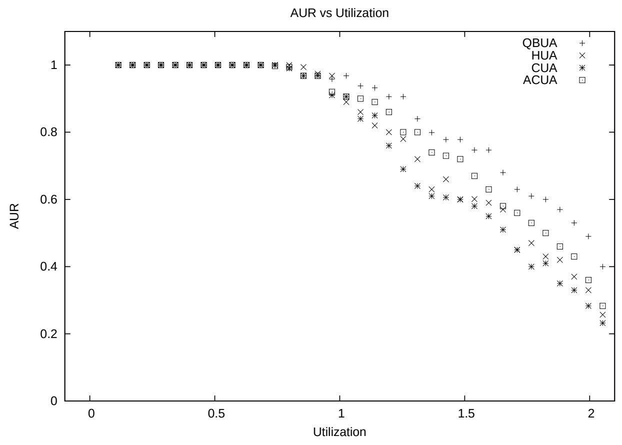

Figure 4.1: QBUA: AUR vs. Utilization (no failures), thread set I

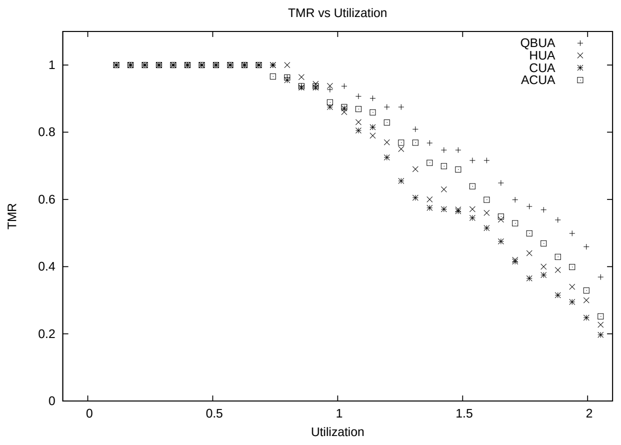

Figure 4.2: QBUA: TMR vs. Utilization (no failures), thread set I

Figures 4.1 and 4.2 show the result of our AUR and DSR experiments in the absence of node failure for thread set I. As Figures 4.1 and 4.2 show, the performance of QBUA during underloads is similar to that of other distributed real-time scheduling algorithms. However, during overloads, QBUA begins to outperform other algorithms due to its better best effort property. During overloads, QBUA accrues, on average, 17% more utility that CUA, 14% more utility than HUA and 8% more utility than ACUA. The maximum difference between the performance of QBUA and the other algorithms in our experiment was the 22% difference between ABUA's and CUA's AUR at the 1.88 system load point. Throughout our experiment, the performance of ACUA was the closest to QBUA with the difference in performance between these two algorithms getting more pronounced as system load increases (the largest difference in performance is 11.7% and occurs at about 2.0 system load). The reason for this behavior is that QBUA has a similar best-effort property to ACUA (see Theorem 34).

In addition, we believe that the difference between these two algorithms becomes more pronounced as system load increases because the delay caused by the scheduling overhead has greater consequences on the schedulability of the system due to the decreased system slack during overloads. Thus QBUA's lower overhead allows it to scale better with system load. Another interesting aspect of the experiment is that, contrary to Theorem 23, QBUA does not accrue 100% utility during all cases of underload. As the load on the system approaches 1.0 some deadlines are missed because the overhead of QBUA becomes more significant at this point. This is also true for other collaborative scheduling algorithms such as CUA and ACUA, and is true to a lesser extent for non-collaborative scheduling algorithms such as HUA due to their lower overhead.

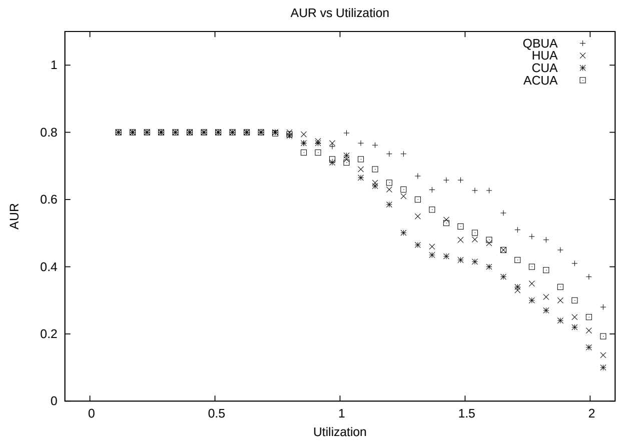

Figure 4.3: QBUA: AUR vs. Utilization (failures), thread set I

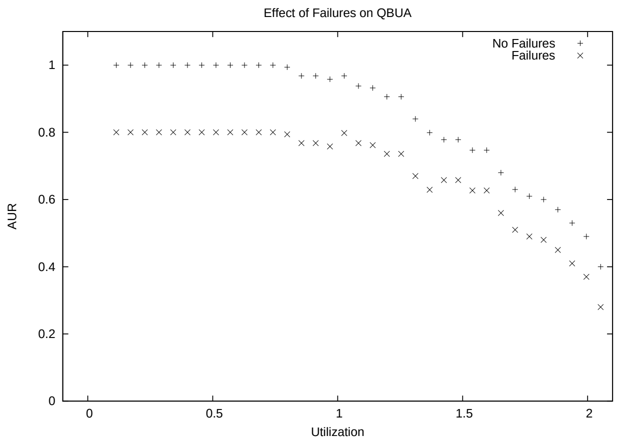

Figure 4.4: QBUA: Effect of failures on QBUA, thread set I

Figures 4.3 and 4.4 show the effect of failures on QBUA when thread set I is used. In these experiments we programmatically fail $f_{max} = 0.2N$ nodes — i.e., we fail 20% of the client nodes. From Figure 4.3, we see that failures do not degrade the performance of QBUA compared to other scheduling algorithms — i.e., the relationship between the utility accrued by QBUA to the utility accrued by other scheduling algorithms remains relatively the same in the presence of failures. However, it is interesting to note that now QBUA accrues, on average, 18.5% more utility than CUA, 13.6% more utility than HUA and 9.9% more utility than ACUA. It should be noticed that both ACUA and CUA suffer a further loss in performance relative to QBUA in the presence of failures. The main reason this occurs is that both these algorithms have a time complexity that is a function of the number of node failures, therefore they have higher overheads in the presence of failures and this affects their results.

In Figure 4.4 we compare the behavior of QBUA in the presence of failure to its behavior in the absence of failure. As can be seen, QBUA's performance suffers a degradation in the presence of failures. As can be seen from the figure, the difference in performance of QBUA in the presence of failure is most pronounced during underloads, and becomes less pronounced as the system load is increased. The reason for this is that during underloads all threads are feasible and therefore the failure of a particular node deprives the system of the utility of all the threads that have a section hosted on that node. However, during overloads, not all sections hosted by a node are feasible, thus the failure of that node only deprives the system of the utility of the feasible threads that have a section hosted by that node. This amount is less than would occur if the system were deprived of the utility of all threads hosted on the node and leads to a less pronounced effect on the performance of QBUA in the presence of failures.

In Figures 4.5, 4.6, 4.7 and 4.8, the same experiments as above are repeated using thread set II. As can be seen, the results follow a similar pattern to the experiments for thread set I, but now the difference between QBUA and other algorithms is not so pronounced. This occurs because the randomly generated thread set is likely to contain some threads that need collaborative scheduling but not as many as in the thread set that we specifically designed to contain such threads.

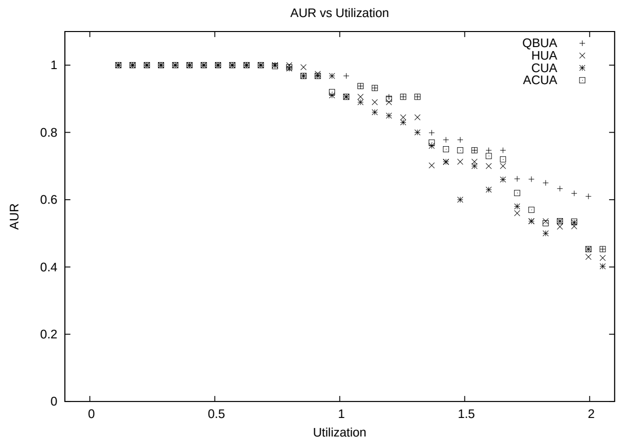

Figure 4.5: QBUA: AUR vs. Utilization (no failures), thread set II

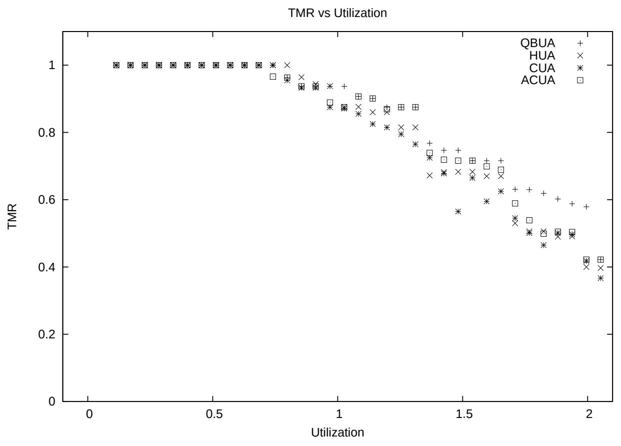

Figure 4.6: QBUA: TMR vs. Utilization (no failures), thread set II

In Figures 4.9, 4.10, 4.11 and 4.12, the same experiments as above are repeated using thread set III. In these experiments, the best performing algorithm is HUA since it has the least overhead (it does not perform any collaborative scheduling). The performance of the other three algorithms are similar to each other.

We also perform a set of experiments to determine the appropriate job time scales to use when QBUA is employed as the scheduling algorithm. For these set of experiments, we fix the communication delay at 60ms. We fixed the execution time for each section (shown on the x-axis of Figure 4.13), and then varied the periods to obtain different utilization factors. We recorded the utilization factor, the Deadline Miss Load (DML), at which the first deadline miss occurred. Theoretically, QBUA should only miss deadlines for overloaded systems, however, practically, the overhead of the system becomes more significant when the section execution time is on the same order as the overhead.

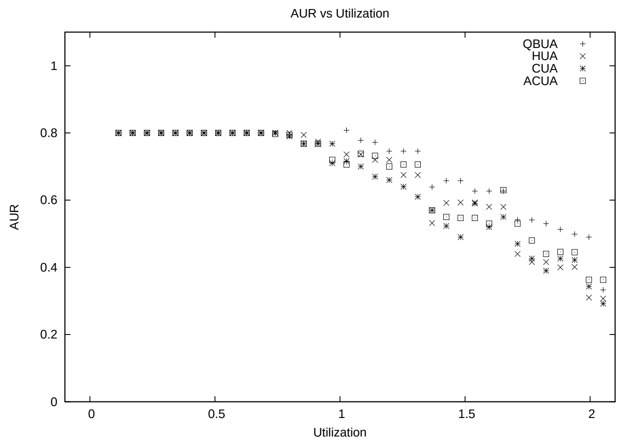

Figure 4.7: QBUA: AUR vs. Utilization (failures), thread set II

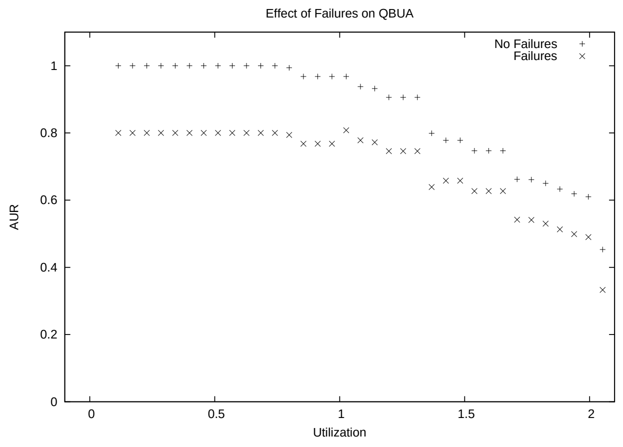

Figure 4.8: QBUA: Effect of failures of QBUA, thread set II

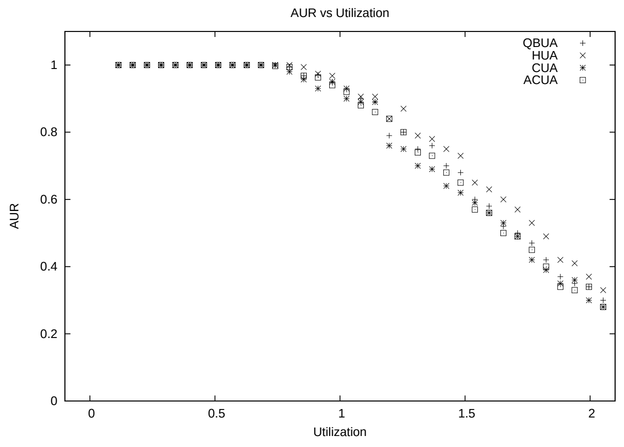

Figure 4.9: QBUA: AUR vs. Utilization (no failures), thread set III

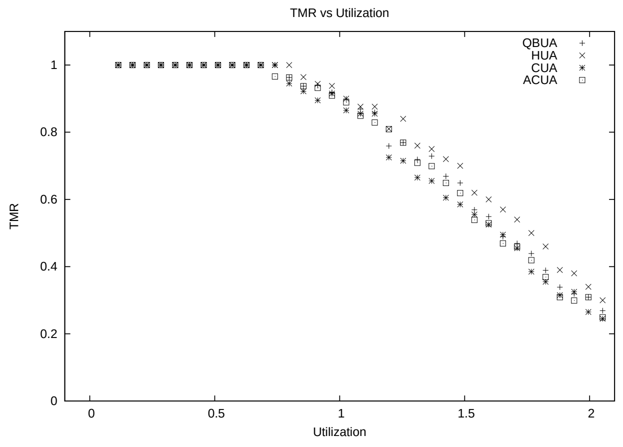

Figure 4.10: QBUA: TMR vs. Utilization (no failures), thread set III

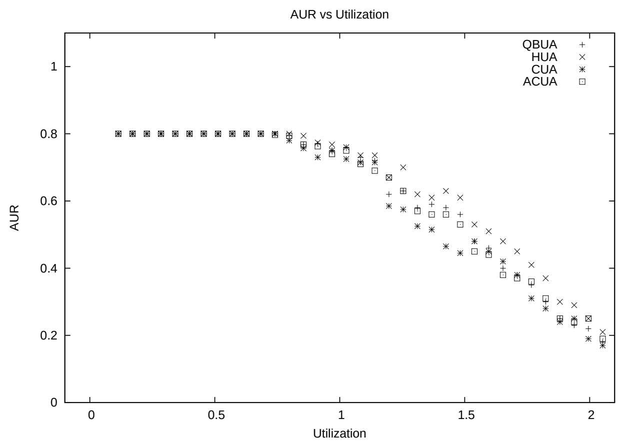

Figure 4.11: QBUA: AUR vs. Utilization (failures), thread set III

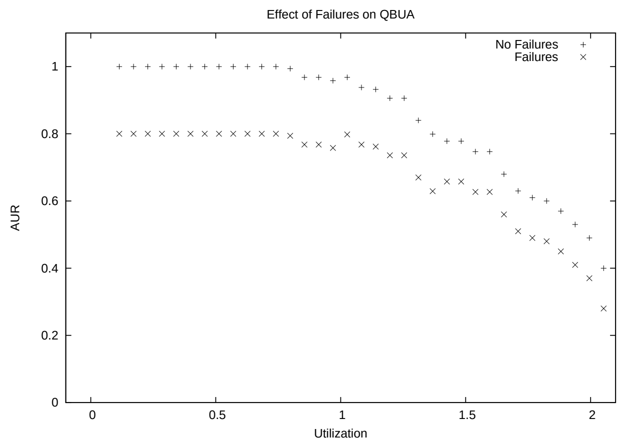

Figure 4.12: QBUA: Effect of failures of QBUA, thread set III

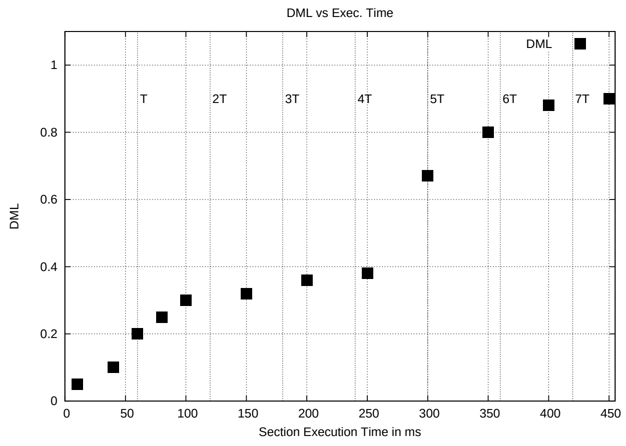

Figure 4.13: QBUA: DML vs. Section Execution Time

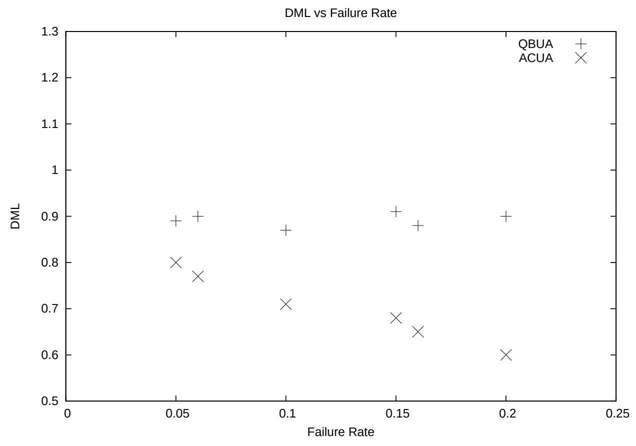

Figure 4.14: QBUA: DML vs. Failure Rate

As can be seen in Figure 4.13, this causes QBUA to perform poorly for systems with small section execution times. However, as the section execution time increases, the DML increases to approach one. In particular, when the section execution times exceeds $5T$ ($2T$ for quorum arbitration without contention) and $3T$ for the algorithm to execute, the system begins to perform adequately.

In Section 4.5, we prove that QBUA scales better than ACUA in the presence of failure. In order to further investigate this, we conducted the following experiment. We fixed the section execution time at $7T$, fixed the failure rate, i.e., the percentage of nodes that fail, (shown on the x-axis of Figure 4.14), and varied the period to determine the utilization, DML, at which the first deadline is missed due to time overruns rather than node failures. Note that this new definition of DML is necessary because we wish to determine the effect of failures on the overhead, thus we need to exclude the deadline misses that occur because nodes hosting certain sections fail.

As can be seen, the DML of ACUA decreases as the failure rate increases, while the DML for QBUA remains relatively unchanged. This occurs because the overhead of ACUA varies in direct proportion to the number of failed nodes, while the overhead of QBUA is independent of the number of failed nodes. This is further elaborated on in Section 4.5.

4.5 QBUA Properties

We now turn our attention to proving some theoretical results for QBUA. Below, T is the communication delay, and Γ is the set of threads in the system.

Lemma 17. *A node determines whether or not it needs to run an instance of QBUA at most* $4T$ *time units after it detects a distributed scheduling event, with high, computable probability,* P_{lock}.

Proof. When a distributed scheduling event is detected by a node, it contacts the quorum system to determine whether or not to start an instance of QBUA (see Section 4.3.2). T time units is used to contact the quorum nodes, and another T time units is taken for the reply of the quorum nodes to reach the requesting node. After these two communication steps, two outcomes are possible.

One possibility is that a quorum ($\frac{2n}{3}$) of servers receive a particular node's request first and so grant that node permission to run an instance of QBUA (lines 15-23, Algorithm 5). In this case, only $2T$ time units are necessary to come to a decision.

The second possibility is that none of the client nodes receive permission from a quorum of server nodes, and therefore the result of the first round of contention is inconclusive (see Section 4.3.2 for an example). In this case, all nodes send YIELD messages to the server nodes to relinquish the "lock" they where granted to run an instance of QBUA (lines 24-30, Algorithm 5). As discussed in Section 4.3.2, the above scenario is caused by differences in communication delays between different client-server pairs. However, by the time that the YIELD messages reach the server nodes ($3T$), all client requests must have reached the server nodes and will be present in their waiting queue (line 12, Algorithm 6) so the server nodes can now make a decision about which node gets to run QBUA (lines 22-31, Algorithm 6) by selecting the earliest request in its waiting queue (ties are broken using client ID). Therefore, the contention is resolved in $4T$ messages delays, one T for each of the REQUEST, RESPONSE, YIELD and RESPONSE messages communicated between client server pairs.

Thus, the whole process of using the quorum system to determine whether or not to run an instance of QBUA takes $4T$ time units in the worst case. Since each of the communication delays, T, are random variables with CDF $DELAY(t)$. The probability that a communication round will take more than T time units is $p = 1 - DELAY(T)$. Since there are four communication rounds, the probability that none of these rounds take more than T time units is $P = bino(0,4,p)$, where $bino(x,n,p)$ is the binomial distribution with parameters n and p. Thus the probability that a node determines whether or not it needs to run a version of QBUA after $4T$ is also $P_{lock} = bino(0,4,p)$. □

Lemma 18. *Once a node is granted permission to run an instance of QBUA, it takes* $O(T + N + |\Gamma|\log(|\Gamma|))$ *time units to compute a new schedule, with high, computable, probability,* P_{SWETS}.

Proof. Once a node is granted permission to run an instance of QBUA it executes Algorithm 4. This algorithm has three communication steps, one to broadcast the START message, another to receive the replies from other nodes in the system and one to multicast any changes to affected nodes. Thus the algorithm takes a total of $3T$ time units for its communication with other nodes.

In addition to these communication steps, Algorithm 4 also takes time to actually compute SWETS (line 3). Algorithm 8 is the algorithm that is used to compute SWETS. In this algorithm, lines 3-7 take $|\Gamma|k$ time units for threads with k sections each. The for loop on lines 8-10 will take wN time units to examine the w sections in the scheduling queue of each of the N nodes in the system. Line 12 takes $O(|\Gamma|\log(|\Gamma|))$ time units to sort the threads in non-increasing order of global PUD using quick sort. The two nested loops on lines 13-26 take $|\Gamma|k^2w$ in the worst case, since there are $|\Gamma|$ threads each with k sections to insert into scheduling queues and each queue needs to be tested for feasibility after the insertion of a section using the linear time function *isFeasible* in $O(w)$ time and removing all previously accepted sections, line 24, can take at most O(k) time. Finally, lines 27-29 determines which nodes need to be notified of changes in $O(N)$ time.

Thus the total time complexity of the algorithm is $3T + |\Gamma|k + wN + O(|\Gamma|\log(|\Gamma|)) + |\Gamma|k^2w + O(N)$. If we consider the number of sections in a thread, k, and the number of sections in the waiting queue of a node, w, to be constants, then the asymptotic time complexity of the algorithm is $O(T+N+|\Gamma|\log(|\Gamma|))$.

There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 4), therefore it is only the third that is probabilistic in nature. Therefore, the probability that SWETS is computed in the time derived above is equal to the probability that the nodes receive the multicast message sent on line 4 of Algorithm 4 within T time units. Since the communication delay has CDF $DELAY(t)$, the probability that T is not violated during runtime, and thus that the time bound above is respected, is $P_{SWETS} = DELAY(T)$. □

%endcomment

Theorem 19. *A distributed scheduling event is handled at most* $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$ *time units after it occurs, with high, computable, probability,* P_{hand}.

Proof. There are two possible distributed scheduling events: 1) the arrival of a new thread into the system and 2) the failure of a node.

In case of the arrival of a new thread, the root node of that thread immediately attempts to acquire a "lock" on running an instance of QBUA. By Lemma 17, the node takes $4T$ time units to acquire a lock and by Lemma 18, it takes the algorithm $O(T+N+|\Gamma|\log(|\Gamma|))$ to compute SWETS. Therefore, in the case of the arrival of a thread the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+4T)$=$O(T+N+|\Gamma|\log(|\Gamma|))$ time units after it occurs. Note that $O(T+N+|\Gamma|\log(|\Gamma|))$ is $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$.

In case of a node failure, some node will detect this failure after T_D time units. That node then attempts to acquire a lock from the quorum system to run an instance of QBUA. By Lemmas 17 and 18, this takes $O(T+N+|\Gamma|\log(|\Gamma|))$ time units. Thus the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$ time units after it occurs.

In both these cases, the result relies on Lemmas 17 and 18, so the probability that events are handled within the time frame mentioned above is $P_{hand} = P_{SWETS} \times P_{lock}$. □

Lemma 20. *The worst case message complexity of the algorithm is* $O(n+N)$.

Proof. The actual message cost of the algorithm is $5n+3N$. The $5n$ component of the message complexity comes from the quorum based arbitration system used in the algorithm. The $5n$ comes from n messages for REQUEST, RESPONSE, YIELD/INQUIRE, RESPONSE and RELEASE respectively. After a node has acquire a "lock", it broadcasts a start message (line 1, Algorithm 4) this takes N messages. The nodes then reply to the current node (line 2, Algorithm 4) using another N messages. Finally, the current node multicasts its results to affected nodes (line 4, Algorithm 4) using another N messages (because in the worst case all nodes in the system may be affected). Thus the actual message complexity of the algorithm is $5n+3N$ which is asymptotically $O(n+N)$. □

Lemma 21. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by QBUA with high, computable, probability* p_{norej}.

Proof. Since the nodes are all underloaded and no nodes fail, Algorithm 8 ensures that all sections will be accepted for scheduling in the system. Therefore, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, $2T$, (see line 2 in Algorithm 4). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 4), that indicates the start of the algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to the node running QBUA during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(T)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast START message reaches all nodes is $P_{tmp} = bino(0,N,p)$ where $bino(x,n,p)$ is the binomial distribution with parameters n and p. If the START message is received, each node sends its schedule to the node that sent the START message. The probability that none of these messages violate the timeout is $tmp = bino(0,N,p)$. As mentioned before, if none of the nodes miss a message, no threads will be rejected, thus the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes send their schedule before the timeout expires. Therefore, $p_{norej} = tmp \times P_{tmp}$. □

Lemma 22. *If each section of a thread meets its derived termination time, then under QBUA, the entire thread meets its termination time with high, computable probability,* p_{suc}.

Proof. Since the termination times derived for sections are a function of communication delay and this communication delay is a random variable with CDF $DELAY(t)$ the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation are violated during runtime.

Let T be the communication delay used in the derivation of section termination times. The probability that T is violated during runtime is $p = 1 - DELAY(T)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above T is $p_{suc} = bino(0,k,p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = bino(0,k,p)$. □

Theorem 23. *If all nodes are underloaded, no nodes fail (i.e.* $f = 0$*) and each thread can be delayed* $O(T + N + |\Gamma|\log(|\Gamma|))$ *time units once and still be schedulable, QBUA meets all the thread termination times yielding optimal total utility with high, computable, probability,* P_{alg}.

Proof. By Lemma 21, no threads will be considered for rejection from a fault free, underloaded system with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 8.

By Lemma 22, this implies that each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

We make the requirement that a thread tolerate a delay of $O(T+N+|\Gamma|\log(|\Gamma|))$ time units and still be schedulable because QBUA takes $O(T+N+|\Gamma|\log(|\Gamma|))$ time units to reach its decision about the schedulability of a newly arrived thread. Thus if this delay causes any of the thread's sections to miss their deadlines, the thread will not be schedulable. We only require that the thread suffer this delay *once* because we assume that there is a scheduling coprocessor on each node, thus the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 24. *If $N-f$ nodes do not crash, are underloaded, and all incoming threads can be delayed $O(T+N+|\Gamma|\log(|\Gamma|))$ and still be schedulable, then QBUA meets the execution time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. As in Lemma 21, no thread in the eligible thread set Γ will be rejected if nodes receive the broadcast START message and respond to that message on time. The probability of these two events is $bino(0,N-f,p)$ where $p = 1-DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bino(0,N-f,p) \times bino(0,N-f,p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 22, this implies that each of these threads, T_j, will meet their termination times with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times if their sections meet their termination times is $P_{tmp} = \prod_{j\in\Gamma} p^j_{suc}$. The probability that all the remaining threads are execute to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. □

Lemma 25. *QBUA has a quorum threshold, m, (see Algorithm 5) of $\lceil \frac{2n}{3} \rceil$ and can tolerate $f^s = \frac{n}{3}$ faulty servers.*

Proof. Our algorithm considers a memoryless crash recovery model for the quorum nodes. This means that a quorum node that crashes and then recovers losses all its state information and starts from scratch. What this implies is that for our algorithm to tolerate such failures, the threshold m should be large enough such that there is at least one correct server in the intersection of any two quorums.

Assume that f is the maximum number of faulty servers in the system (i.e. servers that may fail at some time in the future), then the above requirement can be expresses as $2m-n > f^s$. On the other hand, m cannot be too large since some servers will fail and choosing too large a value of m may mean that client nodes may wait indefinitely for responses from servers that have failed. The requirement translates to $m \leq n - f^s$. Combining the two we get, $f^s = \frac{n}{3}$ and m can be set to $\lceil \frac{2n}{3} \rceil$. □

Definition 3 (Section Failure)**.** *A section, S^i_j, is said to have failed when one or more of the previous head nodes of S^i_j's thread (other than S^i_j's node) has crashed.*

Lemma 26. *If a node hosting a section, S^i_j, of thread T_i fails (per Definition 3) at time t_f, every correct node will include handlers for thread T_i in its schedule by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for QBUA calculated per the analysis in Theorem 19, with high, computable, probability, P_{hand}*

%begincomment

Proof. Since the QoS FD we use in this work detects a failed node in T_D time units [17], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the QBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 15-16 Algorithm 8). Consequently, Algorithm 8 will include the section handlers for this thread in H. Execution of QBUA completes in time t_a and thus all handlers will be included in H by time $t_f + T_D + t_a$.

Of all these timing terms, only t_a is stochastic. From Theorem 19, we know that t_a will be obeyed with probability P_{hand}, therefore, the time bound derived above is also obeyed with probability P_{hand}. □

Lemma 27. *If a section S_i, where $i \neq k$, fails (per Definition 3) at time t_f and section S_{i+1} is correct, then under QBUA, its handler S^h_i will be released no earlier than S^h_{i+1}'s completion and no later than $S^h_{i+1}.tt + T + S^h_i.X - S^h_i.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires (lines 9-10 in Algorithm 10); or 2) an explicit invocation is made by the handler's successor (lines 11-12 in Algorithm 10).

In the first case, we know from the analysis in Section 4.3.2 that the start time of S^h_i is $S^h_{i+1}.tt + S^h_j.X + T - S^h_j.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S^h_j.X \geq S^h_j.ex$ (otherwise the handler would not be schedulable), $S^h_{i+1}.tt + S^h_j.X + T - S^h_j.ex > S^h_{i+1}.tt$, and this satisfies the lower bound of the theorem.

In the second case, an explicit message has arrived indicating the completion of S^h_{i+1}. Since the message was sent, this indicates that $S^h_{i+1}.tt$ has already passed, thus satisfying the lower bound of the theorem. In addition, the message should have arrived T time units after S^h_{i+1} finishes execution (i.e at $S^h_{i+1}.tt + T$), since $S^h_{i+1}.tt + T \leq S^h_{i+1}.tt + T + S^h_i.X - S^h_i.ex$ (remember that $S^h_i.X \geq S^h_i.ex$), then the upper bound is satisfied. □

Lemma 28. *If a section S_i fails (per Definition 3), then under QBUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system wide schedulable set previously constructed. This implies that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$ respectively (lines 18-26 of Algorithm 8).

When some previous head node of S_i's thread fails, QBUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 8 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 8-11 and lines 18-26). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 10), then S_i^h will complete by time $S_i^h.tt$. □

We now state QBUA's bounded clean-up property.

Theorem 29. *In the event of a failure of a thread, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$, with high computable probability P_{exep}.*

Proof. The LIFO property follows from Lemma 27. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 27), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 28, where each correct handler is shown to complete before its termination time.

Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 26) with probability P_{hand} and QBUA guarantees that all these sections will complete before their termination times (Lemma 28). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 4.3.2) is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$ (which is basically the sum of the relative termination times of all the exception handlers, plus the termination time of the last section, which is used as an estimate for the worst case failure time of the threads per the discussion in Section 4.3.2, k communication delays T to notify handlers in LIFO order, T_D to detect the failure after it occurs and t_a for QBUA to execute).

Since Lemma 28 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that QBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 26, we know this probability is P_{hand}, thus $P_{exep} = P_{hand}$. □

Lemma 30. *QBUA has the **DBE** property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(T+N+|\Gamma|\log(|\Gamma|))$ (see Theorem 19) and still be schedulable.*

Proof. The DBE property requires all threads to be ordered in non-decreasing order of global PUD, and this is accomplished in lines 3-7 and line 12 of Algorithm 8. In addition, the DBE property requires that all feasible threads be scheduled in non-decreasing order of PUD, and this is accomplished in lines 13-26 of Algorithm 8. Thus QBUA has the DBE property for all threads that can withstand the $O(T+N+|\Gamma|\log(|\Gamma|))$ overhead of the algorithm and still remain schedulable. □

Lemma 31. *HUA [75], does not have the **DBE** property.*

Lemma 32. *CUA [74] does not have the DBE property.*

Lemma 33. *ACUA has the **DBE** property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(fT+Nk)$ and still be schedulable.*

The proof for Lemmas 31, 32, and 33 can be found in [33].

Theorem 34. *QBUA has a better best-effort property than HUA and CUA and a similar best-effort property to ACUA.*

Proof. The proof follows directly from Lemmas 31, 32, 33 and 30. In particular, HUA and CUA do not have the DBE property while QBUA does, and both QBUA and ACUA have the DBE property but for threads that can survive their, different, scheduling overheads. □

Lemma 35. *The message overhead of QBUA is better than the message overhead of ACUA and scales better with the number of node failures.*

Proof. The message complexity of ACUA is $O(fN^2)$ which is clearly asymptotically more expensive than the O($n+N$) message complexity of QBUA. In addition, since the message complexity of ACUA is a linear function of f, the number of failed nodes, and the message complexity of QBUA does not depend on f —i.e., is not affected by the number of node failures— the message overhead of QBUA scales better in the presence of failure. □

Lemma 36. *The time overhead of QBUA is asymptotically similar to the time overhead of ACUA and scales better with the number of node failures. In addition, when the number of threads in the system is fixed, the time complexity of QBUA is asymptotically better than that of ACUA and scales better in the presence of failure.*

Proof. The time complexities of the two algorithms, $O(T+N+|\Gamma|\log(|\Gamma|))$ for QBUA and $O(fT+kN)$ for ACUA, are asymptotically similar, but since the time complexity of ACUA is a function of f and QBUA's is not, QBUA's time complexity scales better in the presence of failure.

When the number of threads in the system is fixed, the term $|\Gamma|\log(|\Gamma|)$ in the time complexity of QBUA becomes a constant and thus its asymptotic time complexity becomes $O(T+N)$ which is better than the time complexity of ACUA, $O(fT+kN)$. Further, since the time complexity of QBUA is not a function of f and the time complexity of ACUA is, QBUA's complexity scales better in the presence of failure. □

Theorem 37. *QBUA has lower overhead than ACUA and its overhead scales better with the number of node failures.*

Proof. The proof follows directly from Lemmas 35 and 36. □

In should be noted that in our computation of time complexity of algorithms, we do not take into account the effect of message overhead. However, the message complexity affects the utilization of the communication channel and the queue delay at nodes and hence has a direct impact on communication delay (which appears as a term in both time complexities mentioned above). The effect of this higher message complexity can be seen in our experiments (see Section 4.4), where the higher overhead of ACUA, both message and time overheads, results in performance that is worse than that of QBUA.

Theorem 38. *QBUA limits thrashing by reducing the number of instances of QBUA spawned by concurrent distributed scheduling event.*

Proof. Thrashing occurs when concurrent distributed events spawn, superfluous, separate instances of QBUA. QBUA prevents this by having nodes wishing to run an instance of QBUA contact a quorum system to gain permission for doing so (see Section 4.3.2). In lines 9-13 of Algorithm 6, the quorum system does not spawn a new instance of QBUA if there is an instance of QBUA already running that was granted permission to start *after* the timestamp of the arriving scheduling event. This occurs because the instance of QBUA that started after the scheduling event occurred will have information about that event and will thus handle it. This reduces thrashing by prevent superfluous concurrent instances from running at the same time. □

4.6 Conclusion

We presented a collaborative, quorum-based thread scheduling algorithm, QBUA, for unreliable distributed real-time systems. The collaborative approach employed allows QBUA to outperform non-collaborative algorithms during overloads.

This occurs because the collaborative approach allows QBUA to take into account global information when constructing the schedule and thus avoid making locally optimal decisions that can compromise global optimality. The collaborative approach also allows the algorithm to take into account node failures and to attempt to maximize timeliness in the presence of these failures.

In is noticed however, that the higher overhead associated with collaborative approaches allows non-collaborative scheduling algorithms to outperform QBUA during underloads. Despite this disadvantage, QBUA performs better than ACUA, another collaborative scheduling algorithm, since it avoids the use of distributed consensus and instead relies on quorum-based algorithms.

QBUA is designed for a partially synchronous system where message loss and communication delay is stochastically described. Thus QBUA has better coverage than algorithms designed for fully synchronous systems. The performance and properties of QBUA were analytically established and empirically compared to other algorithms.

Chapter 5

Quorum-based-collaborative scheduling with resource dependencies

5.1 Introduction

In this chapter, we consider the problem of scheduling dependent threads in the presence of uncertainties mentioned in Chapter 1. We design a collaborative thread scheduling algorithm, DQBUA, that can handle distributed dependencies. To the best of our knowledge, this is the first collaborative scheduling algorithm to consider distributed dependencies. We compare DQBUA to RTG-DS [37], a dependent thread scheduling algorithm that uses gossip ro improve the reliability of the communication layer and to find the next head node of a thread. RTG-DS falls under the independent category of thread scheduling algorithms.

5.2 Models and Objective

Since this is essentially an extension the algorithm described in Chapter 4 (QBUA), all of the models used there are the same of this algorithm. However, we introduce, below, the resource model which is not used by QBUA.

Resource Model Threads can access serially reusable non-CPU resources (e.g., disks, NICs) located at their nodes during their execution. We consider the single resource model where only one instance of each resource exists in the system. Resources can be shared under mutual exclusion constraints. A thread may request multiple resources during its lifetime but can only have one outstanding request at any given instance of time. Threads explicitly release all granted requests before the end of their execution.

All resource request/release pairs are assumed to be confined within one node. Thus, a node cannot lock a resource on one node and release it on another. However, it is possible for a thread to lock a resource on a node and then make a remote invocation to another node (carrying the lock with it). Since resource reqeust/release pairs are confined to one node, the lock is released when the thread's node returns back to the node on which the resource was acquired.

Resources are assumed to access their resources in an arbitrary order — i.e., which resources are needed by which threads is not known a priori. Consequently we employ deadlock detection and resolution methods instead of prevention and avoidance techniques. Deadlock resolution is performed by aborting one of the deadlocked threads by executing its exception handler.

Scheduling Objectives. Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible in the presence of dependencies. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must bound the time threads remain in deadlock.

5.3 Algorithm Rationale

In [29], we develop QBUA, a scheduling algorithm for distributable real-time threads in partially synchronous systems. In this work, we extend QBUA by adding resource dependencies and precedence constraints handling capabilities, we call the resulting algorithm DQBUA. As in [21], precedence constraints can be programmed as resource dependencies and are handled the same way.

As in QBUA, when a node detects a distributed scheduling event (the failure of a node, the arrival of a new thread into the system or a resource request) it contacts a quorum system requesting permission to run an instance of DQBUA (in order to construct a global schedule). All other scheduling events, such as resource releases and section completion, are dealt with locally, see Algorithm 11. Once permission is granted, it broadcasts a start of algorithm message to all other nodes requesting their scheduling information. Nodes that receive this message reply by sending their scheduling information. When all nodes have sent their scheduling information to the requesting node, it computes a system-wide schedule, which we call a $\underline{\text{S}}$ystem $\underline{\text{W}}$ide $\underline{\text{E}}$xecutable $\underline{\text{T}}$hread $\underline{\text{S}}$et (or SWETS), and multicasts any updates to nodes whose schedule has been affected.

The purpose of the quorum system is to arbitrate among nodes that detect a distributed scheduling event concurrently. This arbitration reduces thrashing by minimizing the number of instances of DQBUA that are started to handle the same or concurrent scheduling events. Due to space limitations, we do not reproduce the details of the quorum arbitration algorithm, see [29] for details.

Algorithm 11: DQBUA: Event Dispatcher on each node i

```
1  Data: schedevent, current schedule $\sigma_p$;
2  switch schedevent do
3      case invocation arrives for $S^i_j$
4          mark segment $S^i_j$ ready;
5      case segment $S^i_j$ completes
6          remove $S^i_j$ from $\sigma_p$;
7          remove $S^h_j$ from $H$;
8          set $RE^i_j$ to zero;
9      case $S^h_j \in H$ and $S^h_i.st$ expires
10         mark handler $S^h_j$ ready;
11     case downstream handler $S^h_{j+1}$ completes
12         mark handler $S^h_j$ ready;
13     case handler $S^h_j$ completes
14         remove $S^h_j$ from $\sigma_p$,$H$;
15         notify scheduler for $S^h_{j-1}$;
16     case new thread, $T_i$, arrives
17         if origin node, send segments $S^i_j$ to all;
18         pass event to DQBUA;
19     case node failure detected
20         pass event to DQBUA;
21     case $S^i_j$ requests a resource
22         pass event to DQBUA;
23     case $S^i_j$ releases a resource
24         free resource;
25 execute first ready segment in $\sigma_p$;
```

While computing a system-wide schedule, threads are ordered in non-increasing order of their global Potential Utility Density (PUD) (which we define as the ratio of a thread's utility to its remaining execution time), the threads are then considered for scheduling in that order. Favoring high global PUD threads allows us to select threads for scheduling that result in the most increase in system utility for the least effort. This heuristic attempts to maximize total accrued utility [21].

Both local and distributed resource dependencies are possible, therefore both local and distributed deadlock can occur. By considering resource requests as distributed scheduling events, DQBUA detects and resolves both local and distributed deadlock in a timely manner. In addition, contention for resources is resolved using their global PUD.

5.4 Algorithm Description

Once the arbitration phase of the algorithm is complete and a node has been granted permission to run an instance of DQBUA, that node runs the algorithm depicted in Algorithm 12. In Algorithm 12, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units for all nodes in the system to respond by sending their local scheduling information (line 2). After collecting this information, the node computes SWETS (line 3) using Algorithm 15. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule).

Algorithm 12: DQBUA: Compute SWETS

```
1: Broadcast start of algorithm message, START;
2: Wait $2T$ collecting replies from other nodes;
3: Construct SWETS using information collected;
4: Multicast change of schedule to affected nodes;
5: return;
```

Algorithm 15 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 3 in Algorithm 12). It performs two basic functions, first, it computes a system wide order on threads by computing their global PUD. It then attempts to insert the remaining sections of each thread, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After the insertion of each thread, the schedule is checked for feasibility. If it is not feasible, then the thread is removed from SWETS (after scheduling the appropriate exception handler if necessary).

First we need to define the global PUD of a thread. Assume that a thread, T_i, has k sections denoted $\{S_1^i, S_2^i, \cdots, S_k^i\}$. We define the global remaining execution time, GE_i, of the thread to be the sum of the remaining execution times of each of the thread's sections. Let $\{RE_1^i, RE_2^i, \cdots, RE_k^i\}$ be the set of remaining execution times of T_i's sections, then $GE_i = \sum_{j=1}^{k} RE_j^i$. Assuming that we are using step-down TUFs, and T_i's TUF is $U_i(t)$, then its global PUD can be computed as: $T_i.PUD = U_i(t_{curr} + GE_i)/GE_i$, where U is the utility of the thread and t_{curr} is the current time. Using global PUD, we can establish a system wide order on the threads in non-increasing order of "return on investment". Thus we consider the threads for scheduling in an order that is designed to maximize accrued utility [21].

In the absence of dependencies, the above computation can be used to represent the utility that would be accrued if a thread where to execute immediately. However, since we consider dependencies, the utility of a thread can only be accrued if all the threads it depends on either complete their execution or are aborted first. Therefore when a section requests a resource, we compute its dependency list by following the chain of resource requests and ownership. Since a resource request is a distributed scheduling event, the node that gets permission to run an instance of DQBUA (after arbitration by the quorum system) will be sent all the information necessary for it to compute the dependency chain.

Once the dependency list has been computed, we compute the PUD of the current thread by using a least effort heuristic —i.e., while examining the threads in the dependency list to compute PUD, if it is faster to abort them than to continue execution, then the threads are aborted and vice versa. Thus we compute the PUD of a thread if it is executed as soon as possible. A similar heuristic is used in [21] but for a single processor, in contrast to the distributed system we consider in this work. Note that this heuristic minimizes the amount of time a high utility thread waits for a resource, at the expense of having to possibly re-execute threads that have been aborted (see [21] for details).

Algorithm 13: DQBUA: computePUD

```
1: Input: $T_i$, $Dep(i,k), j$; //$j$ is node where resource request occurred
2: $Util \leftarrow 0$; $Time \leftarrow 0$; $Seen \leftarrow \emptyset$;
3: for each $Dep(i,k)$ do
4:     for each $S \in Dep(i,k)$ do
5:         if $S.ID \notin Seen$ then
6:             $Seen \leftarrow Seen \cup S.ID$;
7:             //$\Gamma_1$: sections from $S$ until last visit to $j$
8:             $S.Rem \leftarrow \Sigma_{k \in \Gamma_1} RE_k^{S.ID}$;
9:             //$\Gamma_2$: all downstream sections
10:            $S.Abort \leftarrow \Sigma_{k \in \Gamma_2} S_k^h.ex$;
11:            if $S.Abort > S.Rem$ then
12:                $Time \leftarrow Time + S.Rem$;
13:                $Util \leftarrow Util + U_T(t_{curr} + S.Rem)$
14:            else $Time \leftarrow Time + S.Abort$;
15: $Time \leftarrow Time + GE_i$;
16: $Util \leftarrow Util + U_i(t_{curr} + GE_i)$;
17: $T_i.PUD = Util/Time$;
18: return $T_i.PUD$;
```

Since we are computing the PUD of the whole thread, we need to consider the dependencies of all sections belonging to the thread. Therefore, Algorithm 13 considers the dependency list, $Dep(i,k)$, of each of the k sections of thread T_i while computing the PUD. While computing the global PUD of a thread, we take into account the utility of the threads that it depends on. The reason that we do this is that in order to schedule a thread, we need to schedule its dependencies first, so when the thread completes, its dependencies will also have completed thus accruing the utility of both the dependencies and the thread itself. Thus we compute the utility of completing the current thread as the sum of the utility of the current thread and the threads it depends on (lines 13 and 16). We measure the potential utility of a thread and its dependents as the ratio of the utility they can accrue and the time taken for them to accrue this utility (line 17).

We assume that each thread in the system has a globally unique ID, and that each of the thread's sections, S_i, store this global ID in the variable, $S_i.ID$. Since a thread has multiple sections, each of these sections may be dependent on a number of different sections. It is possible that two sections of a thread are dependent on two sections of another thread. In this case, we should only consider the utility of the dependent thread once (since the utility of this thread will only be accrued once when it completes execution). Therefore, in line 5, we check whether a section in a dependency chain, $Dep(i,k)$, belongs to a thread that has been handled before (because another of its sections is in the dependency list of a different section belonging to the current thread). Only threads that have not been considered before are used to compute the current thread's global PUD.

Note that when computing the time remaining for a section, S, to release a resource (line 8), we consider the remaining time for sections starting from S until the last section belonging to S's thread arrives at the current node j. The reason for this is that we do not know when the resource will be released by section S, however since we assume that all resource request/release pairs occur on the same node (see Section 5.2), the latest time at which the resource will be released is when the last section belonging to S's thread to visit node j terminates.

Similarly, when we compute the abort time for section S (line 10), we only consider the abort times of downstream sections. The reason for this is that DQBUA ensures LIFO execution of abort handlers and therefore the current section's abort handler will only execute after the handlers of its downstream sections have terminated. Note that neither the abort time computed in line 10 nor the remaining time to release the resource computed in line 8 are actual times at which those events will occur (interference by other threads ensures that this is not the case), rather the values are merely used as an indication of the amount of work necessary to abort a section to release its resources or to complete a section to release its resources respectively. Since we consider a heuristic of performing the least amount of work, we choose the scenario that takes the least amount of time (line 13).

When computing the global PUD of a section, we need to have up-to-date information about threads that have a section in $Dep(i,k)$. Since resource requests are distributed scheduling events, this information will be received when all nodes in the system send their scheduling information to the node constructing the schedule in response to its broadcast start of algorithm message (lines 1-2 in Algorithm 12).

We now turn our attention to the method used to check schedule feasibility. For a schedule to be feasible, all the sections it contains should complete their execution before their assigned termination time. Since we are considering threads with end-to-end termination times, the termination time of each section needs to be derived from its thread's end-to-end termination time. This derivation should ensure that if all the section termination times are met, then the end-to-end termination time of the thread will also be met. For the last section in a thread, we derive its termination time as simply the termination time of the entire thread. The termination time of the other sections is the latest start time of the section's successor minus the communication delay. Thus the section termination times of a thread T_i, with k sections, is:

$$S^i_j.tt = \begin{cases} T_i.tt & j = k \\ S^i_{j+1}.tt - S^i_{j+1}.ex - T & 1 \le j \le k-1 \end{cases}$$

where $S^i_j.tt$ denotes section S^i_j's termination time, $T_i.tt$ denotes T_i's termination time, and $S^i_j.ex$ denotes the estimated execution time of section S^i_j. The communication delay, which we denote by T above, is a random variable Δ. Therefore, the value of T can only be determined probabilistically. This implies that if each section meets the termination times computed above, the whole thread will meet its termination time with a certain, high, probability. In addition, each section's handler has a **relative** termination time, $S^h_j.X$. However, a handler's **absolute** termination time is relative to the time it is released, more specifically, the **absolute** termination time of a handler is equal to the sum of the **relative** termination time of the handler and the failure time t_f (which cannot be known a priori). To overcome this problem, we delay the execution of the handler as much as possible, thus leaving room for more important threads. We compute the handler termination times as follows:

$$S^h_j.tt = \begin{cases} S^i_k.tt + S^h_j.X + T_D + t_a & j = k \\ S^h_{j+1}.tt + S^h_j.X + T & 1 \le j \le k-1 \end{cases}$$

where $S^h_j.tt$ denotes section handler S^h_j's termination time, $S^h_j.X$ denotes the relative termination time of section handler S^h_j, $S^i_k.tt$ is the termination time of thread i's last section, t_a is a correction factor corresponding to the execution time of the scheduling algorithm, and T_D is the time needed to detect a failure by our QoS FD [17]. From this, we compute latest start times for each handler: $S^h_j.st = S^h_j.tt - S^h_j.ex$ for $1 \le j \le k$, where $S^h_j.ex$ denotes the estimated execution time of section handler S^h_j. Using these derived termination times, we can check whether a schedule is feasible or not.

Algorithm 14 shows how this is done in DQBUA. If the estimated completion time, $S_i.Fin$, of a section is greater than its derived termination, $S_i.tt$, then the schedule is not feasible (lines 13-14). We compute $S_i.Fin$ as the sum of the start time of a section and its execution time. However, it is important to note that, except for current and previous head nodes, these sections haven't arrived at their nodes yet when Algorithm 14 is invoked. Therefore we need to estimate the start time of these sections when computing the estimated completion time of those sections.

We estimate the start time of a section to be the maximum of the estimated completion time of the section preceding it in the local queue (line 10) and the arrival time of the section on a node (which we estimate as the sum of the completion time of the section's predecessor and the communication delay, $S_{i-1}.Fin+T$). We assume that each section's estimated completion time, $S_i.Fin$, is set to infinity before algorithm Algorithm 14 is run.

Algorithm 14: DQBUA: isFeasible

```
1: Input: $\sigma_i$; //Schedule for each node
2: for $1 \leq i \leq N$ do
3:     $pos_i \leftarrow 1$;
4: Until ($pos_i = \text{length}(\sigma_i)$ , $1 \leq i \leq N$) do
5:     for $1 \leq i \leq N$ do
6:         $S_i \leftarrow \text{getElement}(\sigma_i, pos_i)$;
7:         $pre \leftarrow \text{getElement}(\sigma_i, pos_i - 1)$;
8:         if $pos_i = 1$ then $pre.Fin \leftarrow 0$;
9:         if $i = 1$ then $S_{i-1}.Fin \leftarrow S_i.Arr$; $T \leftarrow 0$;
10:        $Start \leftarrow \max(pre.Fin, S_{i-1}.Fin + T)$;
11:        if $Start \neq \infty$ then
12:            $S_i.Fin \leftarrow S_i.ex + Start$;
13:            if $S_i.Fin > S_i.tt$ then
14:                return $false$;
15:            $pos_i \leftarrow pos_i + 1$;
16:    return $true$;
```

We use this relatively expensive method for checking the feasibility of schedules since alternative methods can be misleading. As mentioned before, when Algorithm 14 is invoked, only the current head section and its predecessors will have started in the system, thus the start time of the remaining sections in a thread need to be estimated. The expedient method, used in some previous work, of using the latest start time of a section (computed as its predecessor's latest termination time plus a communication delay, $S_{i-1}.tt+T$) as an estimated for a section's start time means that a section will have no slack time in the schedule. Therefore, the section cannot tolerate any interference by other sections. This usually leads to pessimistic results with some threads being rejected from an underloaded system. Algorithm 14 handles this by computing a better estimate of the start time of sections that haven't started yet.

In Algorithm 15, each node, j, sends the node running DQBUA its current local schedule σ_j^p. Using these schedules, the node can determine the set of threads, Γ, that are currently in the system. Both these variables are inputs to the scheduling algorithm (lines 1 and 2 in Algorithm 15). In lines 3-8, the algorithm DQBUA computes the global PUD of each thread in Γ. The global PUD is computed by first checking whether all sections in a thread can complete execution before their termination time if they were executed immediately. If this is not the case, the thread is assigned a PUD of zero since it cannot possibly accrue any utility to the system (lines 4). Otherwise, we compute the dependency chain for the thread's sections and call Algorithm 13 to compute the global PUD of the thread (lines 6-7). In line 6, we check for cycles to detect any deadlock that may exist. If a cycle is found, it is broken by aborting the thread with the least PUD by executing its exception handler.

Before we schedule the threads, we need to ensure that the exception handlers of any thread

Algorithm 15: DQBUA: ConstructSchedule

```
1: input: $\Gamma$; //Set of threads in the system
2: input: $\sigma_j^p$, $H_j \leftarrow$ nil; //$\sigma_j^p$: Previous schedule of node $j$, $H_j$: set of handlers scheduled
3: for each $T_i \in \Gamma$ do
4:     if for some section $S_j^i \in T_i$, $t_{curr} + S_j^i.ex > S_j^i.tt$ then $T_i.PUD \leftarrow 0$;
5:     else
6:         Compute $Dep(i,j)$, resolving deadlock if necessary;
7:         $T_i.PUD \leftarrow$ ComputePUD($T_i$,$Dep(i,j)$);
8: for each task $el \in \sigma_j^p$ do
9:     if el is an exception handler for section $S_j^i$ then Insert($el$, $H_j$, $el.tt$);
10: $\sigma_j \leftarrow H_j$;
11: $\sigma_{temp} \leftarrow$ sortByPUD($\Gamma$);
12: for each $T_i \in \sigma_{temp}$ do
13:     $T_i.stop \leftarrow false$;
14:     if do not receive $\sigma_j$ from node hosting $S_j^i \in T_i$ then
15:         $T_i.stop \leftarrow true$;
16:     if $T_i.PUD > 0$ and $T_i.stop \neq true$ then
17:         insertByEDF($T_i$,$Dep(i,j)$);
18: for each $j \in N$ do
19:     if $\sigma_j \neq \sigma_j^p$ then Mark node $j$ as being affected;
```

that has already been accepted into the system can execute to completion before its termination time. We do this by inserting the handlers of sections that were part of each node's previous schedule into that node's current schedule (lines 8-9). Since these handlers were part of σ_j^p, and DQBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will meet their termination times.

In line 11, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 12-17). In lines 14-15 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If a thread can contribute non-zero utility to the system and the thread has not been rejected from the system, then we insert its sections, and their dependencies, into the scheduling queue of the node responsible for them in non-decreasing order of termination time by calling Algorithm 16 (lines 16-17).

When Algorithm 16 is invoked, a copy is made of the current schedule so that any changes that result in an infeasible schedule can be undone (line 2). We then consider each of the remaining sections of the thread being considered, if the section does not already belong to the current schedule (because it was part of the dependency chain of a previous thread), the section and its handler are inserted into the current schedule (lines 5-7).

We then consider the dependencies of that section (lines 8-32). Although sections are considered for scheduling in non-increasing order of global PUD, they are inserted into the schedule in non-decreasing termination time order. Thus during underloads, when no threads are rejected, the resulting schedule is basically a deadline ordered list. So during underloads, our scheduling algorithm defaults to Earliest Deadline First (EDF) scheduling, which is an op-

Algorithm 16: DQBUA: insertByEDF

```
1:  input: $\sigma_j^p$, $\sigma_j$;
2:  $\sigma_j^{tmp} \leftarrow \sigma_j$; // make a copy of the schedule
3:  for each remaining section, $S_j^i$, belonging to $T_i$ do
4:      if $S_j^i \notin \sigma_j^{tmp}$ then
5:          Insert($S_j^i$,$\sigma_j^{tmp}$,$S_j^i.tt$);
6:          $TT_{cur} \leftarrow S_j^i.tt$;
7:          if $S_j^h \notin \sigma_j^p$ then Insert($S_j^h$,$\sigma_j^{tmp}$,$S_j^h.tt$);
8:          for $\forall S_n^k \in Dep(i,j)$ do
9:              if $S_n^k \in \sigma_n^{tmp}$ then
10:                 if $S_n^k$ is an abortion handler then
11:                     Remove all sections belonging to $S_n^k$'s thread;
12:                 $TT \leftarrow$ lookUp($S_n^k$,$\sigma_n^{tmp}$);
13:                 if $TT < TT_{cur}$ then
14:                     $TT_{cur} \leftarrow TT$;
15:                     Continue;
16:                 else
17:                     Remove($S_n^k$,$\sigma_n^{tmp}$,$TT$);
18:                     Insert($S_n^k$,$\sigma_n^{tmp}$,$TT_{cur}$);
19:                     $\delta \leftarrow TT - TT_{cur}$;
20:                     for all predecessors, $S_l^x$, of $S_n^k$ do
21:                         //If $S_n^k$ is an abortion handler, $S_l^x$s are also abortion handlers.
22:                         //Otherwise, $S_l^x$s are normal sections
23:                         $TT \leftarrow$ lookUp($S_l^x$,$\sigma_l^{tmp}$); $\gamma \leftarrow \delta$;
24:                         if $S_n^k.tt - TT < \delta$ then
25:                             $\gamma \leftarrow \delta - (S_n^k.tt - TT)$ ;
26:                         Remove($S_l^x$,$\sigma_l^{tmp}$,$TT$);
27:                         Insert($S_l^x$,$\sigma_l^{tmp}$,$TT - \gamma$);
28:             else
29:                 $TT_{cur} \leftarrow \min(TT_{cur}, S_n^k.tt)$;
30:                 Insert($S_n^k$,$\sigma_n^{tmp}$,$TT_{cur}$);
31:                 if $S_n^k$ is not an abortion handler then
32:                     if $S_n^h \notin \sigma_n^p$ then Insert($S_n^h$,$\sigma_n^{tmp}$,$S_n^h.tt$);
33: if isFeasible($\sigma_j^{tmp}$'s)=true then
34:     $\sigma_j \leftarrow \sigma_j^{tmp}$ for all $j$ ;
35: return $\sigma_j$ for all $j$;
```

timal realtime scheduling algorithm [60] that accrues 100% utility during underloads. Note that if a section, S_n^k, in the dependency chain, $Dep(i,j)$, needs to be aborted in order to reduce the blocking time of a thread, then all the sections belonging to S_n^k's thread need to be aborted as well (lines 10-11).

In order to ensure that the order of the dependencies is maintained, if the termination time of a section is greater than the termination time of a section that depends on it, its termination time is moved up to the termination time of the section that depends on it (lines 17 and 27). In addition, all the predecessors of that current section have their termination time adjusted to reflect this new value (lines 20-27).

5.5 Algorithm Properties

We now turn our attention to proving some theoretical results for the algorithm. Below, T is the communication delay, Γ is the set of threads in the system and k is the maximum number of sections in a thread.

Lemma 39. *A node determines whether or not it needs to run an instance of DQBUA at most $4T$ time units after it detects a distributed scheduling event, with high, computable probability, P_{lock}.*

Proof. This follows from Lemma 1 in [29]. □

Lemma 40. *Once a node is granted permission to run an instance of DQBUA, it takes $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ time units to compute a new schedule, with high, computable, probability, P_{SWETS}.*

Proof. Once a node is granted permission to run an instance of DQBUA it executes Algorithm 12. This algorithm has three communication steps, one to broadcast the START message, another to receive the replies from other nodes in the system and one to multicast any changes to affected nodes. Thus the algorithm takes a total of $3T$ time units for its communication with other nodes. In addition to these communication steps, Algorithm 12 also takes time to actually compute SWETS (line 3). Algorithm 15 is the algorithm that is used to compute SWETS. In this algorithm, lines 3-7 iterate $|\Gamma|$ times, and the function computePUD, invoked in line 7, takes $|\Gamma|k^2$ time in the worst case. Therefore the time complexity of lines 3-7 is $|\Gamma|^2k^2$.

Lines 8-9 take $O(|\Gamma|k)$ time in the worst case, line 11 sorts the threads in $O(|\Gamma|\log(|\Gamma|))$ time. The for loop in lines 12-17 iterates $|\Gamma|$ times in the worst case. The body of this loop calls Algorithm 16 in line 17. The time complexity of Algorithm 16 is dominated by three nested loops in lines 3-32. The outer loop iterates k times in the worst case, the middle loop (starting at line 8) iterates $O(|\Gamma|k)$ times in the worst case, and the inner most loop (starting at line 20) iterates k times in the worst case. The time complexity of the body of the inner loop is dominated by the time complexity of the peek, insert and remove operations (lines 23, 26 and 27 respectively). Using self-balancing binary search trees to represent the queues, all these operations can be performed in $O(\log(|\Gamma|k))$ time. Therefore the nested loop structure, and thus Algorithm 16, has a time complexity of $O(|\Gamma|k^3\log(|\Gamma|k))$. Therefore the time complexity of lines 12-17 is $O(|\Gamma|^2k^3\log(|\Gamma|k))$.

Thus the complexity of the algorithm is $O(|\Gamma|^2k^2+|\Gamma|k+|\Gamma|\log(|\Gamma|)+|\Gamma|^2k^3\log(|\Gamma|k))$, which is asymptotically $O(|\Gamma|^2k^3\log(|\Gamma|k))$. Adding the communication delay to this computational complexity we get $O(|\Gamma|^2k^3\log(|\Gamma|k)+3T)$, which is asymptotically $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$. There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 12), therefore it is only the third that is probabilistic in nature. Therefore, the probability that SWETS is computed in the time derived above is equal to the probability that the nodes receive the multicast message sent on line 4 of Algorithm 12 within T time units. Since the communication delay has CDF $DELAY(t)$, the probability that T is not violated during runtime, and thus that the time bound above is respected, is $P_{SWETS}=DELAY(T)$. □

Theorem 41. *A distributed scheduling event is handled at most* $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$ *time units after it occurs, with high, computable, probability,* P_{hand}.

Proof. There are three possible distributed scheduling events: 1) the arrival of a new thread into the system, 2) a resource request and 3) the failure of a node.

In case of the arrival of a new thread or a resource request, the head node of the thread immediately attempts to acquire a "lock" on running an instance of DQBUA. By Lemma 39, the node takes $4T$ time units to acquire a lock and by Lemma 40, it takes the algorithm $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ to compute SWETS. Therefore, in the case of the arrival of a thread or a resource request the event is handled $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+4T)$=$O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ time units after it occurs. Note that $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ is $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$.

In case of a node failure, some node will detect this failure after T_D time units. That node then attempts to acquire a lock from the quorum system to run an instance of DQBUA. By Lemmas 39 and 40, this takes $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ time units. Thus the event is handled $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$ time units after it occurs.

In both these cases, the result relies on Lemmas 39 and 40, so the probability that events are handled within the time frame mentioned above is $P_{hand}=P_{SWETS}\times P_{lock}$. □

Lemma 42. *If all nodes are underloaded and no nodes fail, then no threads will be suggested for rejection by DQBUA with high, computable, probability* p_{norej}.

Proof. Since the nodes are all underloaded and no nodes fail, Algorithm 15 ensures that all sections will be accepted for scheduling in the system. Therefore, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, $2T$, (see line 2 in Algorithm 12). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 12), that indicates the start of the algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to the node running DQBUA during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(T)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast START message reaches all nodes is $P_{tmp} = bino(0,N,p)$ where $bino(x,n,p)$ is the binomial distribution with parameters n and p. If the START message is received, each node sends its schedule to the node that sent the START message. The probability that none of these messages violate the timeout is $tmp = bino(0,N,p)$. As mentioned before, if none of the nodes miss a message, no threads will be rejected, thus the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes send their schedule before the timeout expires. Therefore, $p_{norej} = tmp \times P_{tmp}$. □

Lemma 43. *If each section of a thread meets its derived termination time, then under DQBUA, the entire thread meets its termination time with high, computable probability, p_{suc}.*

Proof. Since the termination times derived for sections are a function of communication delay and this communication delay is a random variable with CDF $DELAY(t)$ the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation are violated during runtime.

Let T be the communication delay used in the derivation of section termination times. The probability that T is violated during runtime is $p = 1 - DELAY(T)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above T is $p_{suc} = bino(0,k,p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = bino(0,k,p)$. □

Theorem 44. *If all nodes are underloaded, no nodes fail (i.e. $f = 0$) and each thread can be delayed $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units once and still be schedulable, DQBUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg}.*

Proof. By Lemma 42, no threads will be considered for rejection from a fault free, underloaded system with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 15.

By Lemma 43, this implies that each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p_{suc}^{j}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

We make the requirement that a thread tolerate a delay of $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units and still be schedulable because DQBUA takes $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units to reach its decision about the schedulability of a newly arrived thread. Thus if this delay causes any of the thread's sections to miss their deadlines, the thread will not be schedulable. We only require that the thread suffer this delay *once* because we assume that there is a scheduling coprocessor on each node, thus the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 45. *If $N - f$ nodes do not crash, are underloaded, and all incoming threads can be delayed $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ and still be schedulable, then DQBUA meets the termination time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. As in Lemma 42, no thread in the eligible thread set Γ will be rejected if nodes receive the broadcast START message and respond to that message on time. The probability of these two events is $bino(0, N - f, p)$ where $p = 1 - DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bino(0, N - f, p) \times bino(0, N - f, p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 43, this implies that each of these threads, T_j, will meet their termination times with probability p_{suc}^{j}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times if their sections meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p_{suc}^{j}$. The probability that all the remaining threads are execute to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. The reason for tolerating $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ delay is the same as in Theorem 44. □

Definition 4 (Section Failure)**.** *A section, S_j^i, is said to have failed when one or more of the previous head nodes of S_j^i's thread (other than S_j^i's node) has crashed.*

Lemma 46. *If a node hosting a section, S_j^i, of thread T_i fails (per Definition 4) at time t_f, every correct node will include handlers for thread T_i in its schedule by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for DQBUA calculated per the analysis in Theorem 41, with high, computable, probability, P_{hand}*

Proof. Since the QoS FD we use in this work detects a failed node in T_D time units [17], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the DQBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 14-15 Algorithm 15). Consequently, Algorithm 15 will include the section handlers for this thread in the schedule. Execution of DQBUA completes in time t_a and thus all handlers will be included in the schedule by time $t_f + T_D + t_a$.

Of all these timing terms, only t_a is stochastic. From Theorem 41, we know that t_a will be obeyed with probability P_{hand}, therefore, the time bound derived above is also obeyed with probability P_{hand}. □

Lemma 47. *If a section S_i, where $i \neq k$, fails (per Definition 4) at time t_f and section S_{i+1} is correct, then under DQBUA, its handler S_i^h will be released no earlier than S_{i+1}^h's completion and no later than $S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires; or 2) an explicit invocation is made by the handler's successor.

In the first case, we know from the analysis in Section 5.4 that the start time of S_i^h is $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S_j^h.X \geq S_j^h.ex$ (otherwise the handler would not be schedulable), $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex > S_{i+1}^h.tt$, and this satisfies the lower bound of the theorem.

In the second case, an explicit message has arrived indicating the completion of S_{i+1}^h. Since the message was sent, this indicates that $S_{i+1}^h.tt$ has already passed, thus satisfying the lower bound of the theorem. In addition, the message should have arrived T time units after S_{i+1}^h finishes execution (i.e at $S_{i+1}^h.tt + T$), since $S_{i+1}^h.tt + T \leq S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$ (remember that $S_i^h.X \geq S_i^h.ex$), then the upper bound is satisfied. □

An interesting thing about the property above is that it is not probabilistic in nature. At first sight, it would seem that the property is stochastic due to the probabilistic communication delay used in the second case mentioned in the proof. One would expect the upper bound in the property to be respected only probabilistically in the second case. However, if the upper bound is not met in the second case (i.e. the stochastic communication delay causes the notification of the completion of handler S_{i+1}^h to arrive after the upper bound in the theorem), then the first case kicks in and starts the handler before the upper bound expires anyway. Therefore this result is deterministic in nature.

Lemma 48. *If a section S_i fails (per Definition 4), then under DQBUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system wide schedulable set previously constructed. This implies that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$ respectively (lines 5-7 of Algorithm 16).

When some previous head node of S_i's thread fails, DQBUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 15 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 8-9 and line 10). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 7, Algorithm 16), then S_i^h will complete by time $S_i^h.tt$. □

Note that the termination times mentioned in the proofs above may be modified to reflect dependencies (lines 20-27 in Algorithm 16). We now state DQBUA's bounded clean-up property.

Theorem 49. *In the event of a failure of a thread, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$, with high computable probability P_{exep}.*

Proof. The LIFO property follows from Lemma 47. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 47), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 48, where each correct handler is shown to complete before its termination time.

Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 46) with probability P_{hand} and DQBUA guarantees that all these sections will complete before their termination times (Lemma 48). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 5.4) is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$ (which is basically the sum of the relative termination times of all the exception handlers, plus the termination time of the last section, which is used as an estimate for the worst case failure time of the threads per the discussion in Section 5.4, k communication delays T to notify handlers in LIFO order, T_D to detect the failure after it occurs and t_a for DQBUA to execute).

Since Lemma 48 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that DQBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 46, we know this probability is P_{hand}, thus $P_{exep} = P_{hand}$. □

Theorem 50. *A deadlock is resolved in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units by terminating the thread that can contribute the least amount of utility to the system.*

Proof. A resource request is a distributed scheduling event. Therefore, when the resource request that causes a wait-for cycle to form occurs, it is handled in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units (see Theorem 41). While computing the dependency chain in Algorithm 15, the cycle will be detected and broken by terminating the thread in the cycle with the lowest PUD. The theorem follows. □

Theorem 51. *Resource contention is resolved in order of thread PUD.*

Proof. Threads are ordered according to their PUD in Algorithm 15. Therefore if more than one thread is waiting for a particular resource, threads with higher PUD will be considered before threads with lower PUD. □

Theorem 52. *DQBUA limits thrashing by reducing the number of instances of DQBUA spawned by concurrent distributed scheduling event.*

Proof. This follows from the proof of Theorem 22 of [29]. □

5.6 Experimental Results

We performed a series of simulation experiments on ns-2 to compare the performance of DQBUA to RTG-DS in terms of Accrued Utility Ratio (AUR) and Deadline Satisfaction Ratio (DSR). We define AUR as the ratio of the accrued utility (the sum of U_i for all completed threads) to the utility available (the sum of U_i for all available jobs) and DSR as the ratio of the number of threads that meet their termination time to the total number of threads in the system. We considered threads with three segments. Each thread starts at its origin node with its first segment. The second segment is a result of a remote invocation to some node in the system, and the third segment occurs when the thread returns to its origin node to complete its execution.

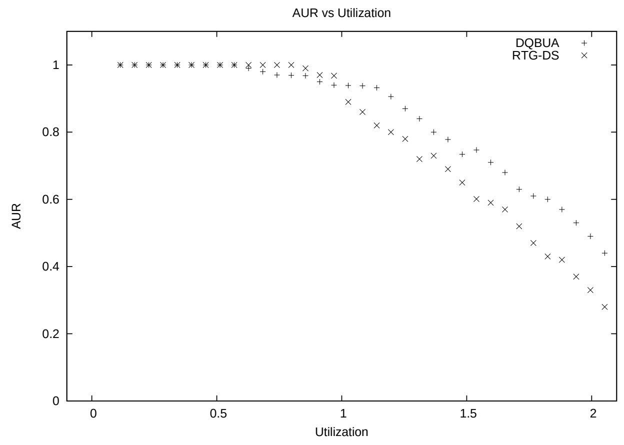

Figure 5.1: DQBUA: AUR vs. Utilization

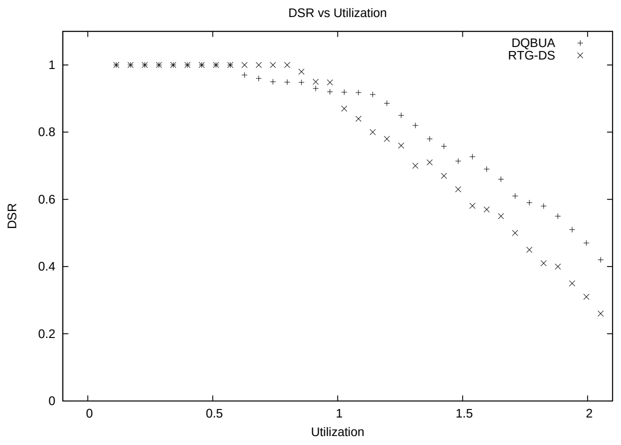

Figure 5.2: DQBUA: DSR vs. Utilization

The periods of these threads are fixed, and we vary their execution times to obtain a range of utilization ranging from 0 to 200%. For fair comparison, all algorithms were simulated using a synchronous system model, where communication delay varied according to an exponential distribution with mean and standard deviation 0.02 seconds but could not exceed an upper bound of 0.5 seconds. Our system consisted of fifty client nodes and five servers. In our experiments, the utilization of the system is considered the *maximum* utilization experienced by any node. We assume that there are two, different, resources on each node in the system. Each section randomly choose which resource, if any, it wishes to acquire. The time spent using a resource is a uniformly distributed random number that represents a proportion of that section's remaining execution time.

DQBUA is a collaborative scheduling algorithm, as such, its strength lies in its ability to give priority to threads that will result in the most system-wide accrued utility even if the sections of those threads do not maximize local utility on the nodes they are hosted. The thread set that highlights this property contains threads that would be given low priority if local scheduling is performed but should be assigned high priority due to the system-wide utility they accrue. Therefore, we chose a thread set that contains high utility threads that have one section with above average execution time (resulting in low PUD for that section) and other sections with below average execution times (resulting in high PUD for those sections). Such thread sets test the ability of the algorithm to take advantage of collaboration to avoid making locally optimal decisions that would compromise global optimality.

As can be seen in Figures 5.1 and 5.2, the performance of DQBUA is better than that of DTG-DS during overloads. This occurs, because DQBUA performs collaborative scheduling thus maximizing, as much as possible, **system-wide** accrued utility. On the other hand, RTG-DS does not perform collaborative scheduling (but uses gossip to identify the next head node of a thread and to improve the reliability of the communication layer) and therefore performs worse during overloads.

5.7 Conclusions

We presented an algorithm, DQBUA, for scheduling dependent distributable threads in a partially synchronous system. We showed that it accrues optimal utility during underloads and attempts to maximize the accrued utility during overloads. We experimentally compared DQBUA to another scheduling algorithm for dependent threads, RTG-DS, and showed that DQBUA outperforms RTG-DS during overloads.

Chapter 6

The case for STM

6.1 Introduction

Recently, due to fundamental physical constraints such as heat emanations, the computer industry has undergone a paradigm shift: increasing computer performance is now done by increasing the number of cores on a chip rather than increasing clock speed [86]. Today, most machines produced are multi-core and the use of distributed systems is on the increase. Coinciding with this new direction of using concurrency to increase application throughput, is the discovery of a rich set of applications that are a natural fit for parallel and distributed architectures. From distributed databases to emerging distributed real-time systems [12], such emerging applications are only meaningful in a distributed system with multiple computing cores cooperating to execute the semantics of the application.

This parallelism offers a great opportunity for improving performance by increasing application concurrency. Unfortunately, this concurrency comes at a cost: programmers now need to design programs, using existing operating system and programming language features, to deal with shared access to serially reusable resources and program synchronization. The de facto standard for programming such systems is using threads, locks, and condition variables. Using these abstractions, programmers have been trying to write correct concurrent code ever since multitasking operating systems made such programs possible.

Unfortunately, the human brain does not seem to be well suited for reasoning about concurrency [58]. The history of the software industry contains numerous cases where the difficulty inherent in reasoning about concurrent code has resulted in costly software errors that are very difficult to reproduce and hence debug and fix. Among the more common errors encountered in lock-based software systems are deadlocks, livelocks, lock convoying, and, in systems where priority is important (e.g, embedded real-time systems), priority inversion. Such errors stem from the difficulty in reasoning about concurrent code.

Transactions have proven themselves to be a successful abstraction for handling concurrency in database systems. Due to this success, researchers have attempted to take advantage of their features for non-database systems. In particular, there has been significant recent efforts to apply the concepts of transactions to shared memory. Such an attempt originated as a purely hardware solution [42, 54] and was later extended to deal with systems where transactional support was migrated from the hardware domain to the software domain [80]. Software transactional memory (or STM) has, until recently, been an academic curiosity because of its high overhead. However, as the state-of-the-art improved and more efficient algorithms were devised, a number of commercial and non-commercial STM systems have been developed (see implementations section of [90]). In this chapter, we discuss the issues involved in implementing software transactional memory in distributed embedded real-time systems.

6.2 Motivation

Currently, the industry standard abstractions for programming distributed embedded systems include OMG/Real-Time CORBA's client/server paradigm and distributable threads [67] and OMG/DDS's publish/subscribe abstraction [71]. The client/server and distributable threads abstractions directly facilitate the programming of causally-dependent, multi-node application logic. In contrast, the publish/subscribe abstraction is a data distribution service for logically-single hop communications (i.e., from one publisher to one subscriber), and therefore, higher-level abstractions must be constructed – on an application-specific basis – to express causally-dependent, multi-node application logic (e.g., publication of topic A depends on subscription of topic B; B's publication, in turn, depends on subscription of topic C, and so on). All of these abstractions rely on lock-based mechanisms for concurrency control, and thus suffer from their previously mentioned inherent limitations.

In particular, lock-based concurrency control can easily result in local and distributed deadlocks, due to programming errors that occur as a result of the conceptual difficulty of the (lock-based) programming model. Detecting and resolving deadlocks, especially distributed deadlocks, that can potentially arise due to distributed dependencies is complex and expensive. Note that deadlocks can only be detected and resolved, as opposed to being avoided or prevented, in those distributed embedded systems where it is difficult to obtain a-priori knowledge of which activities need which resources and in what order. When a deadlock is detected in such systems, the usual method of resolving it is to break the cycle of the waiting processes by terminating one of them. Unfortunately, the choice of which process to terminate is not a simple one in real-time systems. By terminating one of the processes that are waiting in a cycle, we produce a chain of waiting processes. Depending on how, i.e., where, we break this cycle, it may or may not be feasible to meet the timing requirements of the remaining processes. Thus, we need to consider the structure of the dependency chain, after terminating a process to end the deadlock, in order to break the cycle in a way that optimizes end-to-end timeliness objectives. Furthermore, a process's dependencies must be taken into account when making the choice about which process to terminate. For example, if a significant number of processes depend on the result of a process, terminating it to resolve a deadlock may not be in the best interest of the application. In addition, the cost of deadlock detection/resolution is exacerbated by the extra work necessary to restore the system to an acceptable state when failure occurs. Thus, deadlock resolution is a complex process.

The problem of distributed deadlock detection and resolution has been exhaustively studied, e.g., [24, 25, 27, 57, 65, 76, 81]. A number of these algorithms turned out to be incorrect by either detecting phantom deadlocks (false positives) or not detecting deadlocks when they do exist, e.g., [19, 25]. These errors occur because of the inherent difficulty of reasoning about distributed programs. This led to attempts at providing a formal method for analyzing such protocols to ensure correct behavior (e.g., [24]). Despite the difficulty of reasoning about distributed deadlock, solutions for this problem on synchronous distributed systems have been developed. Unfortunately, for asynchronous systems, errors in the deadlock detection process become inevitable. For real-time systems, these issues become more severe [81]. The semantic difficulty of thread and lock based concurrency control and the high overhead associated with detecting and resolving distributed deadlock, as indicated above, are the driving motivations for finding different programming abstractions for distributed embedded real-time systems. Chapter 2 contains a review of the literature regarding this matter and contains our reasoning for believing that STM is a promising solution to this problem.

6.3 STM for distributed embedded systems

There are a number of competing abstractions for implementing STM in distributed embedded real-time systems. An interesting abstraction is the notion of real-time distributed transactional objects, where code is immobile and objects migrate between nodes to provide a transactional memory abstraction. Another alternative is to allow remote invocations to occur within a transaction, spawning sub-transactions on each node (where they are executed using STM), and using a distributed commit protocol to ensure atomicity. A third alternative is to provide a hybrid model, where both data and code are mobile and the decision of which is moved is heuristically decided either dynamically or statically. Several key issues need to be studied in order to use STM in distributed embedded systems, these are:

- Choosing an appropriate abstraction for including STMs in distributed embedded systems;
- Designing the necessary protocols and algorithms to support these abstractions;
- Implementing these abstractions in a programming language by making necessary changes to its syntax and in the run-time environment; and

- Designing scheduling algorithms to provide end-to-end timeliness using these new programming abstractions.

6.3.1 Choosing an appropriate abstraction.

STM is a technology for multiprocessor systems, to use it in a multicomputer environment, we need to develop appropriate abstractions. We are currently considering three competing programming abstractions into which to incorporate STM:

- A model where cross-node transactions are permitted using remote invocations and atomicity is enforced using an atomic commit protocol;
- A model where a distributed cache coherence protocol is used to implement an abstraction of shared memory on top of which we can build STM; and
- A hybrid model where code or data is migrated depending on a number of heuristics such as size and locality.

In the first approach, we manage concurrency control on each node using STM, but allow remote invocations to occur within a transaction. Thus we allow a transaction to span multiple nodes. At the conclusion of the transaction, the last node on which transactional code is executed acts as a coordinator in a distributed commit protocol to ensure an atomic commitment decision. Our preliminary research, which we intend to elaborate upon, indicates that such an approach may be prone to "retry thrashing" especially when the STM implemented on each node is lock-free.

Since lock-free STM is an optimistic concurrency control mechanism, extending the duration of a transaction by allowing it to sequentially extend across nodes results in a significantly higher probability of conflicts among transactions. Such conflicts lead to aborted transactions that are later retried. Retrying is antagonistic to real-time systems since it degrades one of the most important features of real-time systems: predictability. Lock-based STM tends to reduce some of this "thrashing" behavior since it eliminates part of the "optimism" of the approach. However, long transactions are still more susceptible to retries and introducing locks into the STM implementation necessitates a deadlock detection and resolution solution. Fortunately such a solution does not need to be distributed since it only needs to resolve local deadlocks.

Implementing STM on top of a distributed cache coherence protocol has been investigated in [43, 61]. In this approach, code is immobile, but data objects move among nodes as required. The approach uses a distributed cache coherence protocol to find and move objects. We intend to design real-time cache coherence protocols, where timeliness is an integral part of the algorithm. We plan to design STM on top of these protocols and compare their performance to the flow control abstraction. An important advantage of this approach is that it eliminates the need for a distributed commit protocol. Since distributed commit protocols are a major source of inefficiency in real-time systems [35], such an approach is expected to yield better performance.

The last approach we intend to study is touched upon in [10]. This is a hybrid approach where either data objects or code can migrate while still retaining the semantics of STM. By allowing either code or data to migrate, we can choose a migration scenario that results in the least amount of communication overhead. For example, suppose we have a simple transactional program that increments the value of a shared variable X and stores the new value in the transactional store. Assume further that X is remote, using a data flow abstraction would necessitate two communication delays; one to fetch X from its remote location and the other to send it back once it has been incremented. Using a control flow abstraction in this case may be more efficient since it will only involve a single communication delay.

On the other hand, assume that several processes need access to a small data structure and that these processes are in roughly the same location and are far away from the data they need. Since communication delay depends on distances, it may make sense to migrate the data to the processes in this case rather than incur several long communication delays by moving the code to the data. In short, the choice of whether to migrate code or data can have a significant effect on performance. In [10], this is accomplished under programmer control by allowing an *on* construct which a programmer can use to demarcate code that should be migrated. We intend to elaborate on this by coming up with solutions that would use static analysis at compile-time (or dynamically at run-time) to make decisions about which part of the application to move using a number of heuristics such as, for example, size of code/data and locality considerations.

6.3.2 Designing suitable protocols and algorithms.

The algorithms and protocols that need to be designed depend on the programming abstraction we choose to implement. Some of the necessary abstractions have been touched upon in Section 6.3.1, here we elaborate on these points.

For the model where code migrates, creating cross-node transactions, and data is immobile, the main abstraction that needs to be designed is a real-time distributed commit protocol. Since cross-node transactions are permitted, with each node involved hosting part of the transaction, a distributed commit protocol is necessary to ensure atomicity. A number of distributed commit protocols have been studied in the literature, with the two phase commit protocol being the most commercially successful protocol. Unfortunately, the blocking semantics of the two phase commit protocol may not be very suitable for real-time systems. Therefore alternatives like the three phase commit protocol (despite its larger overhead) may be more appropriate due to its non-blocking semantics. Other alternatives that involve the relaxation of certain properties of distributed commit protocols in order to improve efficiency are discussed in [35]. We intend to design distributed commit protocols whose timeliness behavior can be quantified theoretically and/or empirically, in order to allow the system to provide guarantees on end-to-end timeliness.

For the approach where code is immobile and data migrates, the most important protocol that needs to be designed is a distributed real-time cache coherence protocol. This protocol needs to be location aware in order to reduce communication latency and should be designed to reduce network congestion. The cache coherence problem for multiprocessors has been extensively studied in the literature [83]. There are also some solutions for the distributed cache coherence problem (see [1, 15, 52, 87] for a, not necessarily representative, sample of research on this issue). Distributed cache coherence bears some similarity to distributed hash table (or DHT) protocols which have been an active topic of research recently due to the popularity of peer-to-peer applications. Examples of DHT algorithms that are of interest are [45, 73, 77].

We envision a cache coherence algorithm based on hierarchical clustering to reduce network traffic and path reversal to synchronize concurrent requests, an approach used in [43]. Other approaches for implementing distributed cache coherence will also be considered. An important part of our research in this area will be to design cache coherence protocols that can provide timeliness guarantees that we can verify theoretically and empirically.

For the hybrid abstraction, where both code and data can move, several issues need to be determined. Among the issues that need to be resolved are the different methods of distributing transactional meta-data in order to ensure efficient execution of the STM system, providing a mechanism to support atomic commitment when code is allowed to migrate thus resulting in multi-node transactions, aggregating communication in order to reduce the effect of the extra communication necessary to manage the STM system (possibly by piggybacking this information over normal network traffic) and optimizing network communication to reduce latency. It is also necessary to design appropriate mechanisms for choosing whether data or code migration is going to occur. Currently, the choice of which part of the program to migrate is performed under programmer control [10]. We intend to design automated methods for deciding which part of the program moves through either compile-time analysis or at run-time.

6.3.3 Programming language implementation.

We need to incorporate the programming abstraction chosen and the protocols and algorithms necessary to support them into a suitable programming language. Issues that need to be addressed are extending the programming language syntax to include support for higher level abstractions built upon STM. We introduce a number of syntactic modifications to support the new constructs we propose to implement. The most basic syntactic extension required is a method for demarcating atomic blocks (i.e. blocks of code that will be executed within the context of STM), additions such as programmer controlled retry and providing alternative transactional execution can also be considered.

In addition to these syntactic extensions, modifications to the run-time environment are also required. Our top candidate for implementing these abstractions is the emerging DRTSJ RI. We choose this language for a number of reasons. First, the language is still under development with a substantial part of the implementation details coming out of our research group. Second, the RI will be evaluated by the standard's expert community (e.g., Sun's JSR-50 experts group in the case of DRTSJ) as part of the standard's approval process, resulting in immediate and invaluable user feedback. Third, using a garbage collected language alleviates some of the issues involved in memory management associated with STM (by, for example, eliminating the problem of having transactions free allocated memory explicitly while other transactions are still working on it). Of course this necessitates augmenting the garbage collector with information about STM in order to prevent harmful interference with STM's meta-data.

Naturally, the actual modifications made to the programming language will depend on the programming abstraction chosen. Regardless of the choice made about the abstraction used to incorporate STM in distributed embedded systems, modifications to the run-time environment are necessary to support STM. The actual modifications made are dependent on the particular design we choose for our implementation of STM and so will not be elaborated upon in this chapter. However, some of the design issues involved are choosing appropriate meta-data to represent STM objects, providing appropriate mechanisms to atomically commit transactions (for example by using atomic hardware instructions such as compare-and-swap, or CAS, on suitably indirected meta-data), providing implementations for the different design choices of STM (e.g., visible reads versus invisible reads and weak versus strong atomicity).

6.3.4 Scheduling algorithms and analysis.

Finally, we will design scheduling algorithms that allow systems programmed using STM to meet end-to-end timeliness requirements. This is a challenge due to the fact that the retry behavior of STM is antagonistic to predictability. There have been several attempts at providing timing assurances when STM is used in real-time systems or when lock-free data structures are used in real-time systems [3–5, 62]. These approaches only consider uni-processor systems and use the periodic task arrival model to bound retries.

Some of the approaches are fairly sophisticated and use, for example, linear programming [3] to derive schedulability criteria for lock-free code. The basic idea of these approaches is that, on a uni-processor system, the number of retries is bounded by the number of task preemptions that occur. This bound exists because a uni-processor can only execute one process at a time. Since it is not possible for a process to perform conflicting operations on shared memory, and hence cause the retry of another process, unless it is running, the number of preemptions naturally bounds the number of retries on uni-processors. Given this premise, the analysis performed in [3–5, 62] bounds the number of retries by bounding the number of times a process can be preempted under different scheduling algorithms. This analysis allows the authors to derive schedulability criteria for different scheduling algorithms based on information about process execution times, execution times of the retried code sections, process periods, etc.

More recently, attempts have been made at providing timeliness guarantees for lock-free data structures built on multiprocessor systems [47]. The approach used in [47] is suitable for Pfair-scheduled systems and other multiprocessor systems where quantum-based scheduling is employed. The most restrictive assumption made in this approach is that access to a shared lock-free object takes at most two quanta of processor time. Using this assumption, the authors go on to bound the number of retries by determining the worst-case number of accesses that can occur to a shared object during the quanta in which it is being accessed. For an $\boldsymbol{M}$ processor system, the worst-case number of processes that can interfere with access to a particular shared object is $\boldsymbol{M}-1$. Given an upper bound on the number of times a process can access a shared object within a quanta, it is possible to derive an upper bound on the number of retries in such a system. The authors also go on to describe how it is possible to use the concept of a "supertask", basically a single unit that is composed of several tasks that are to be scheduled as one unit, to reduce the worst-case number of retries and hence improve system performance.

The particular method used to bound the number of retries in the system(s) we develop will depend on the model we target. There are two possible alternatives. The first approach is to target uni-processor distributed systems. In such systems, each node has only one processor. In order to provide scheduling criteria for such systems, we would use the approaches developed for uni-processor systems to derive the number of retries that can occur on each node, and then combine these bounds to determine the number of retries that can occur to cross-node transactions, thus deriving schedulability criteria for STM implementations.

The second approach is to consider multiprocessor distributed systems. In such systems, each node is a multiprocessor or multi-core machine. Schedulability analysis and scheduling algorithms for such systems are considerably more difficult due to the difficulty in deriving bounds on the number of retries in the system. A first possible approach is to consider the Pfair-scheduling algorithm considered in [47] for obtaining bounds on the number of retries on each node and then combining these bounds to obtain bounds for cross-node transactions. Other approaches will also be considered in order to reduce the number of assumptions made on the system model. We will design scheduling algorithms that can ensure that timeliness requirements are not violated by the retry behavior of STM on distributed systems, and provide analytical expressions for the schedulability criteria of these scheduling algorithms.

6.4 Conclusions

Programming distributed systems using lock-based concurrency control is semantically difficult and computationally expensive. In order to alleviate some of these problems, we propose the use of STM for concurrency control. In order to achieve this goal, a number of issues need to be addressed. This chapter outlines these issues and proposes a method for solving them.

Three different abstractions for incorporating STM into distributed embedded real-time systems are mentioned, and the algorithms and protocols necessary for implementing these abstractions are briefly outlined. We also briefly indicate the type of schedulability analysis that will be required to provide timeliness guarantees for systems programmed using these abstractions.

Chapter 7

Response time analysis of uni-processor distributed systems using STM

7.1 Introduction

As mentioned in Chapter 6, STM is a promising alternative to traditional lock-based concurrency control. However, while STM has many promising features, it is not a silver bullet. Some problems associated with STM include handling irrevocable instructions such as I/O, the weak atomic semantics of some of the current implementations [31] and the overhead of retries. Despite these disadvantages, its semantic simplicity makes it a very promising alternative to lock-based concurrency control.

In this chapter, we consider using STM as the concurrency control mechanism for (non-I/O code in) distributed real-time systems. Toward this, we propose a method for computing an upper bound on the worst-case response time of periodic tasks, programmed using STM, running in a distributed real-time system that employs Earliest Deadline First (EDF) scheduling.

7.2 General Framework

There are different ways of incorporating STM into distributed systems [31]. In this chapter, we consider a model where a distributed application consists of several pieces of code, which we will refer to as tasks, executing on single nodes that are subject to crash failures. Concurrency control on each node is managed using STM. A task makes an invocation (a procedure call or an RPC, depending on whether the successor resides on the same node) after it has finished execution (at which point all its STM transactions should have been committed). We do not allow cross-node critical sections (i.e. cross node transactions).

We present a method for analyzing the worst-case response times in such a system. In order to do this, we first show how to extend Spuri's response time analysis technique [82] to include the overhead associated with the retry behavior of STM on a single machine without considering offsets in Section 7.3.2. We then extend this analysis to include offsets in Section 7.4.2. Including offsets is necessary since it enables us to handle the precedence constraints of a distributed system where tasks make RPC calls to remote nodes and therefore some tasks cannot start before their predecessor makes an invocation.

The ability to use offsets and jitters to represent these precedence constraints is discussed in Section 7.6. In Section 7.5.1, we show how failures can be considered in the analysis by ensuring that exception handlers can be executed if necessary. Our goal is to prove that it is possible to provide real-time assurances for distributed systems where concurrency control is managed using STM. Introducing STM to the repertoire of programming tools available for the real-time programmer can considerably improve the quality of distributed concurrent real-time programs and reduce the software development time by reducing the complexity of the programming environment. As such, this chapter is our first step towards achieving the goal of studying STM for distributed real-time systems in all its possible varieties as outlined in [31].

7.3 Tasks with jitter

Given a set of periodic, independent tasks scheduled by EDF on a single processor, Spuri, in [82], proposed an algorithm for computing an upper bound on the worst-case response time for a task. In this section we extend the algorithm to consider tasks with mutually exclusive resource access requirements that are programmed using software transactional memory. We extend Spuri's analysis to consider these tasks and compare the tightness of the bound we obtain by comparing it to the utilization based schedulability analysis of lock-free code proposed in [4].

Spuri's idea for computing an upper bound on worst-case response time is based on finding a "critical instant", the release time of a task, τ_a, that would cause it to experience maximal interference from other tasks. This critical instant is found in a busy period, which is defined as a period of time during which a processor is busy executing tasks in the system. The following theorem is helpful in finding that critical instant:

Theorem 53 (Spuri [82]). *The worst-case response time of a task τ_a is found in a busy period in which all other tasks are released simultaneously at the beginning of the busy period, after having experienced their maximum jitter.*

The proof of this theorem, presented in [82], rests on the fact that the conditions in Theorem 53 result in the largest number of interferences from other tasks. The same condition holds for tasks using STM, since the largest number of interferences will result in the largest number of transaction retries and hence the worst-case response time. Note that we make the assumption that each interference by a task results in a transactional retry. This is a pessimistic assumption since the task may not be executing its transactional code at this point in time, or the transactional operations may not conflict before the transaction commits. However, we make this assumption to give our analysis the nature of an upper bound. In Section 7.3.2, we present the modified analysis for independent tasks on a single processor.

7.3.1 Task model

We consider a task model with periodic tasks scheduled with the EDF discipline. The system is composed of a set of N periodic tasks executing in a single processor. Each task τ_i is activated periodically with a period of T_i, has a computation time of $C_i + m_i s$, where C_i is the computation time of the task instance without considering the transactional part of the code, m_i is the number of times transactional code is executed in the task activation and s is the computation cost of the transactional part of the code. In the rest of the chapter, we shall refer to a task activation as *job*. Each job has a relative deadline d_i and a release jitter bounded by J_i. We refer to the absolute deadline of a job as D_i.

7.3.2 Analysis

In this section, we compute the worst-case contribution of a task τ_i to the response time of the task under analysis τ_a. Specifically, we compute the worst-case contribution of task τ_i during a busy period of duration t when the deadline of τ_a is D. Without loss of generality, we label the time at which the busy period starts as t_0, and measure the duration of the busy period, t, and the deadline of τ_a, D, from t_0.

As per Theorem 53, the worst-case contribution of a task, τ_i, occurs when it is released at the start of the busy period after having experienced its maximum jitter. This scenario is chosen so as to maximize the number of instances of τ_i that occur during the busy period in order to maximize interference.

Only jobs with a deadline less than or equal to D can interfere with τ_a, also, jobs that start outside the busy period, t, do not contribute to the interferences that occur within that period. Using this information, we can compute the maximum number of jobs of τ_i that can interfere with τ_a.

From [69], we know that the number of jobs of τ_i within the busy period, p_t, is:

$$p_t = \left\lceil \frac{t + J_i}{T_i} \right\rceil \tag{7.1}$$

and the number of task instances with deadlines at or before D is:

$$p_D = \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \tag{7.2}$$

Since both conditions, p_D and p_t, must be satisfied, the maximum number of interferences of the jobs of τ_i in τ_a is:

$$n_i = min\left(\left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \tag{7.3}$$

The zero that appears as a subscript in the equation above causes the result of the bracketed expression to be zero if its value is negative. This is necessary because if $d_i > D$, no activation of τ_i will interfere with τ_a. Thus, the worst-case contribution of τ_i to the response time of τ_a is:

$$W_i(t, D) = n_i(C_i + m_i s) \tag{7.4}$$

Given this equation, it is possible to compute the response time of τ_a after having determined the critical instant. Unfortunately, we do not know which instant in the busy period is the critical instant. However, it is known that the critical instant can be found either at the beginning of the busy period, or at an instant of time such that the deadline of the analyzed job of τ_a coincides with the deadline of a task τ_i's job. Otherwise, it would be possible to make the activation time of τ_a earlier, without changing the schedule, to increase the response time. The set of instants, Ψ, at which the deadline of τ_a's job coincides with the deadline of the job of some other task in the busy period, is:

$$\Psi = \bigcup \{(p-1)T_i - J_i + d_i\} \tag{7.5}$$
$$\forall p = 1 \cdots \left\lceil \frac{L + J_i}{T_i} \right\rceil, \ \forall i$$

In the above equation, L is the worst-case length of a busy period. The following recurrence relation can be used in computing L:

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil C_i^{exe} \tag{7.6}$$

where C_i^{exe} is an estimate of the processing load placed on the processor by a job during the busy period.

Equation (7.6), one of the many recurrence equations that can be found in response time analysis [53], can be solved by starting with a small initial value for L and then iterating until the equation converges. The equation is guaranteed to converge if the system is not over-utilized (i.e., the load is under 100%). The execution time of a job, without considering any interference, is $C_i + m_i s$. To this execution time, we need to add the load that a job can place on the processor during the busy period if it interferes with another job. For uni-processor systems scheduled using EDF an interference can only occur when a new job arrives with a lower deadline than those already executing causing a context switch. Thus, a job can cause, at most, one retry in some other job. Therefore, the load a job places on the processor during the busy period is $C_i + m_i s + s$, giving us:

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil (C_i + m_i s + s) \tag{7.7}$$

Thus, we can compute the set of critical instants by subtracting d_a from each element in Ψ. We then consider all the critical instants in our analysis to find the critical instant that gives us the worst-case response time.

There may be several jobs of τ_a in the busy period, therefore we need to examine all of these jobs in order to determine which one of them results in the worst-case response time. Assuming that the first instance of τ_a occurs A time units after the start of the busy period, the completion time of job p of τ_a, $w_a^A(p)$, can be computed as:

$$w_a^A(p) = p(C_a + m_a s) + \sum_{\forall i \neq a} W_i(w_a^A(p), D^A(p)) + Is \tag{7.8}$$

where I is the maximum number of interferences that can occur in jobs of higher priority (lower deadline) than the instant of τ_a being studied as expressed in Equation (7.9).

$$I = \sum_{\forall i} min\left(\left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \tag{7.9}$$

In Equation (7.8), the term $D^A(p)$ is the deadline of job p when the first job of τ_a occurs A time units after the start of the busy period and can be computed as:

$$D^A(p) = A - J_a + (p-1)T_a + d_a \tag{7.10}$$

The response time is obtained by subtracting the activation time from the completion time of each task:

$$R_a^A(p) = w_a^A(p) - A + J_a - (p-1)T_a \tag{7.11}$$

Next we need to determine the set of values we will use for A. Note that we have already established that the critical instant can only be in the set Ψ computed in Equation (7.5). We now further narrow down the set of values that need to be considered. Naturally, for each value of p we only need to check values of A within one period. Thus, we can further narrow down the critical instants we should consider to:

$$\Psi^* = \{\Psi_x \in \Psi \mid (p-1)T_a - J_a + d_a \leq \Psi_x < pT_a - J_a + d_a\} \tag{7.12}$$

For each of the values, Ψ_x, in Ψ^*, we need to check the values $A(\Psi_x) = \Psi_x - [(p-1)T_a - J_a + d_a]$.

Finally, we determine the worst-case response time by examining all instants of τ_a within the busy period and taking the maximum response time as our result:

$$\begin{gathered} R_a = \max R_a^A(p) \\ \forall p = 1 \cdots \left\lceil \frac{L-J_a}{T_a} \right\rceil, \forall A(\Psi_x) \mid \Psi_x \in \Psi^* \end{gathered} \tag{7.13}$$

In Sections 7.4.2 and 7.6, this analysis is extended to allow us to handled distributed systems (while Section 7.5.1 shows how we can include exception handlers in the analysis).

7.4 Tasks with jitter and offsets

In this section, we study a system where groups of tasks are grouped together into logical entities which we shall call "transactions". Each transaction is activated by a periodic external event. Once this external event has arrived, the tasks in each transaction begin execution after a certain time period, which we refer to as the offset, has passed. This model can be used to model distributed systems. Already, several papers have been published showing how to obtain an upper bound on the response time of distributed systems programmed using this task model, e.g., [38, 69, 72]. Those attempts are based on the Holistic analysis first proposed by Tindell and Clark [89]. In this section, we extend such analysis to deal with STM based concurrency control.

7.4.1 Task model

We consider a system composed of a set of tasks executing on the same processor. These tasks are grouped into logical entities referred to as transactions. Each transaction, Γ_i, is activated by a periodic external event with period T_i. A transaction consists of n_i tasks (not to be confused with the n_i used in the analysis of Section 7.3.2). We designate the tasks τ_{ij}, with the first subscript, i, identifying the transaction the task belongs to, and the second subscript, j, specifying the order of the task within the transaction in non-decreasing order of offsets.

Each of these tasks has an execution time of $C_{ij} + m_{ij}s$, where C_{ij} and s are the execution times of the transactional and non-transactional part of a task, respectively, and m_{ij} is the number of times transactional code is invoked in a task. Tasks are activated after a certain time, which we shall refer to as the offset, elapses from the arrival of the external event that triggered the transaction. For each τ_{ij}, its offset is designated ϕ_{ij}. We also allow a task to suffer release jitter which is bounded by the term J_{ij}. Both offsets and jitter can be larger than the period of their transaction.

7.4.2 Analysis

In this section, we compute the worst-case response time for task τ_{ab}. In order to do this, we must determine the worst-case contribution of each transaction to the response time of the task under analysis. The theorem below can be used for this purpose:

Theorem 54 (Palencia and Harbour [69]). *The worst-case contribution of transaction Γ_i to the response time of a task τ_{ab} is obtained when the first activation of some task τ_{ik} that occurs within the busy period coincides with the beginning of the busy period, after having experienced the maximum possible delay, i.e., the maximum jitter, J_{ik}.*

Again, this theorem is based on the fact that the worst-case contribution will occur when the most number of tasks are released within the busy period. For the sake of being concise, we will not reproduce the whole derivation of the analysis, which can be found in [69], but will only include the final results and the modifications necessary for accommodating STM.

The worst-case contribution of a task, τ_{ij}, to the response time of the task under analysis, τ_{ab}, during a busy period of duration t and deadline D, when the task whose activation time coincides with the start of the busy period is τ_{ik}, is:

$$W_{ijk}(t,D) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil, \left\lfloor \frac{D - \varphi_{ijk} - d_{ij}}{T_i} + 1 \right\rfloor \right) \right)_0 (C_{ij} + S) \tag{7.14}$$

where $\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) \mod T_i$. Thus, the contribution of transaction Γ_i is the summation of the contribution of all its tasks:

$$W_{ik}(t,D) = \sum W_{ijk}, \quad \forall j \in \Gamma_i \tag{7.15}$$

In order to make the analysis tractable, the worst-case contribution of a transaction, Γ_i, is considered to be the maximum of all possible contributions that could have been caused by considering each of the tasks of Γ_i as the start of the busy period:

$$W_i^*(t,D) = \max\left(W_{ik}(t,D)\right), \quad \forall k \in \Gamma_i \tag{7.16}$$

We number the activations of a job within the busy period using the index p, and consider the first activation to start within the busy period to have an index of $p = 1$. The activations that start before the busy period and suffer their maximum jitter to start at the beginning of the busy period have indices $p \leq 0$. Thus, we can determine the index of the first, $P_{0,ijk}$, and last, $P_{L,ijk}$, activations to contribute to the busy period as follows:

$$P_{0,ijk} = -\left\lfloor \frac{J_{ij}+\varphi_{ijk}}{T_i} \right\rfloor + 1 \qquad (7.17)\ ,P_{L,ijk} = \left\lceil \frac{L-\varphi_{ijk}}{T_i} \right\rceil \qquad (7.18)$$

where L is the maximum length of the busy period as computed in Equation (7.7).

Thus, like in Section 7.3.2, the set of values to analyze is:

$$\Psi = \bigcup \{\varphi_{ijk} + (p-1)T_i + d_{ij}\} \tag{7.19}$$
$$\forall p = P_{0,ijk} \cdots P_{L,ijk}, \quad \forall j,k \in \Gamma_i$$

Thus, if the first activation of τ_{ab} occurs after A time units from the start of the busy period, the worst-case completion time of activation p of task τ_{ab} can be computed as:

$$W^A_{abc}(p) = (p - P_{0,ijk} + 1)(C_{ab} + s) \tag{7.20}$$
$$+W^-_{ac}(W^A_{abc}(p), D^A_{abc}(p)) + \sum_{\forall i \neq a} W_i(W^A_{abc}(p), D^A_{abc}(p))$$

where W^-_{ac} is the result of Equation (7.15) without considering the contribution of τ_{ab} and $D^A_{abc}(p)$ is the deadline of activation p when the first one occurs at time A:

$$D^A_{abc}(p) = A + \varphi_{abc} + (p-1)T_a + d_{ab} \tag{7.21}$$

Thus, we can obtain the response time of a task by subtracting from the completion time the arrival time of the external event:

$$R^A_{abc}(p) = W^A_{abc}(p) - A - \varphi_{abc} - (p-1)T_a + \phi_{ab} \tag{7.22}$$

Also, as in Section 7.3.2, we only need to check the value of A within one period, thus, the points we need to check are:

$$\Psi^* = \{\Psi_x \in \Psi \mid \varphi_{abc} + (p-1)T_a + d_{ab} \leq \Psi_x < \varphi_{abc} + pT_a + d_{ab}\} \tag{7.23}$$

For each value of Ψ_x above, we check $A = \Psi_x - [\varphi_{ijk} + (p-1)T_a + d_{ab}]$. Naturally, the worst-case response time is the maximum response time obtained from the analysis, i.e.,

$$R_{ab} = \max(R^A_{abc}(p)) \tag{7.24}$$
$$\forall p = P_{0,abc} \cdots P_{L,abc} \quad \forall c \in \Gamma_A, \quad \forall A \in \Psi^*$$

7.5 Handling Failures

In this section, we extend our analysis to take failures into account. In some distributed systems, failures are the norm rather than the exception. Therefore, it is necessary to provide some form of assurance on system performance in their presence. We assume that each task, τ_{ij}, has an exception handler that can be used to restore the system to a safe state in case of failure, and that this exception handler has an execution time C^h_{ij} and relative deadline d^h_{ij}. The absolute deadline of the handler is relative to the time that failure is detected, t_f, i.e., $D^h_{ij} = t_f + d^h_{ij}$.

When a node fails, all the jobs executing on that node cease to exist. Since we are considering "transactions" where a sequence of consecutive jobs execute within one logical computational context, it is necessary to understand the effect of failures on this abstraction. Naturally, a failure may fragment a transaction leading to several orphan jobs (i.e., jobs that have been disconnected from their downstream predecessor due to node failure). These jobs need to be identified and their exception handlers need to be executed in order to restore the system to a safe state.

Therefore, in order to have a fault-tolerant system, it must be possible to execute the exception handlers before their deadlines when failure occurs. In this section, we show how we can take the execution time of the exception handlers into account when computing response times in order to ensure safe execution of the system in the presence of failures.

7.5.1 Analysis

We need to determine the maximum number of exception handlers that can execute within a busy period in order to take their overhead into account. As in Sections 7.3.2 and 7.4.2, there are two conditions that determine the number of jobs, in this case exception handlers, that can contribute to the worst-case response time of an instance of a task, τ_{ab}, within busy period of duration t; 1) The number of exception handlers that execute within the duration of the busy period (including exception handlers that were released before the busy period but whose activation time is delayed until the start of the busy period), and 2) the number of handlers with deadline less than or equal to the deadline of the job being analyzed.

From [69], we know that the number of interferences from the jobs of task τ_{ij} that occur from jobs that start at the beginning of a busy period, after suffering some jitter, is:

$$x_i = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{7.25}$$

Each of these activations has an associated exception handler. In the worst-case, each job executes to completion, thus, using up as much processor time as possible, and then an error occurs that causes the trigger of its exception handler, thus, using up more processor time to execute the handler. It is this worst-case scenario from which we derive the deadline of the exception handlers. Note that since we are considering activations of the same task, and all of these activations start at the beginning of the busy period, their exception handlers have the same deadline which is:

$$D_{ij}^{h_{first}} = d_{ij} + d_{ij}^{h} \tag{7.26}$$

If we consider, without loss of generality, that the beginning of the busy period t_B is time zero, the contribution of these exception handlers is $x_i C_{ij}^{h}$ if $D_{ij}^{h_{first}} \leq D$ and zero otherwise.

We now turn our attention to determining the number of interferences that occur from job instances started within the busy period. Below, is the equation that determines the number of instances that can occur within a busy period of duration t:

$$n_{inst} = \left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil \tag{7.27}$$

For each of these activations, the deadline of their handler can be computed as:

$$\begin{gathered} De = \varphi_{ijk} + (p-1)T_i + d_{ij} + d_{ij}^{h} \\ \forall p = 1 \cdots n_{inst} \end{gathered} \tag{7.28}$$

and they each contribute a factor of C_{ij}^{h} to the worst-case response time if their deadline is less than or equal to D.

At this point we have computed the contribution of the execution time of handlers of the activations of τ_{ij} that start at or after the beginning of the busy period t_B. Figure 7.1 depicts the types of scenarios we will be considering. The term in Equation (7.25) represents the

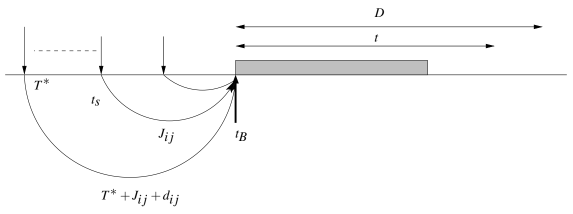

Figure 7.1: Uniprocessor: Scenario for calculating worst-case contribution

number of jobs that can be delayed at most J_{ij} so that their activation starts at t_B. In Figure 7.1, the first such job starts at t_s; other jobs that follow it will be delayed an amount of time less than J_{ij} in order to start at the beginning of the busy period. Equation (7.27) computes the number of jobs that will start after the beginning of the busy period, i.e., after t_B. However, if failures are not considered, jobs that start before t_s, such as the one depicted as starting at time T^* in Figure 7.1, do not contribute to the busy period. This lack of contribution occurs because even if these jobs were delayed J_{ij}, their activation time would fall before t_s. However, now that we consider failures, it is possible for these tasks to contribute to the worst-case response time if their exception handlers start within the busy period. Using similar reasoning as in Section 7.4.2, we compute the worst-case contribution of these exception handlers by considering how many can start at the beginning of the busy period. The latest start time of a handler whose job starts at T^* is:

$$S = T^* + d_{ij} \tag{7.29}$$

If this start time, S, is greater than or equal to t_B then the handler will contribute to the busy period. In other words:

$$T^* + d_{ij} \geq T^* + nT_i + J_{ij} \tag{7.30}$$

which gives us:

$$n < \frac{d_{ij} - J_{ij}}{T_i} \tag{7.31}$$

Since n is an integer, Equation (7.31) resolves to:

$$n = \left(\left\lceil \frac{d_{ij} - J_{ij}}{T_i} \right\rceil - 1 \right)_0 \tag{7.32}$$

As before, the zero subscript indicates that negative values are considered zero (in this case, such an event indicates that none of the jobs starting before t_s can contribute to the busy period). Thus, the contribution of these jobs is nC_{ij}^h if $d_{ij}^h \leq D$ and zero otherwise. We can now compute the contribution of the exception handlers of τ_{ij} to the response time of a job of τ_{ab} in a busy period of duration t and deadline D, when the task whose activation time coincides with the start of the busy period is τ_{ik}, using the following function:

Thus, we can modify Equation (7.14) from Section 7.4.2 to:

$$\begin{aligned}&W_{ijk}(t,D) = \\ &\left(\left\lfloor \frac{J_{ij}+\varphi_{ijk}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t-\varphi_{ijk}}{T_i} \right\rceil, \left\lfloor \frac{D-\varphi_{ijk}-d_{ij}}{T_i} + 1 \right\rfloor\right)\right)_0 (C_{ij}+S) \\ &+ W^h_{ijk}(t,D)\end{aligned} \tag{7.33}$$

Also, we need to modify the critical instants to be examined in order to accommodate the inclusion of the exception handlers in the busy period, thus, Equation (7.19) becomes:

$$\Psi = \bigcup\{\varphi_{ijk} + (p-1)T_i + d_{ij}\} \bigcup \{\varphi_{ijk} + (p-1)T_i + d_{ij} + d^h_{ij}\} \tag{7.34}$$
$$\forall p = P_{0,ijk} \cdots P_{L,ijk}, \quad \forall j,k \in \Gamma_i$$

Algorithm 17: $W^h_{ijk}(t,D)$

```
1: sum=0;
2: if $d^h_{ij} \le D$ then
3:     $n = \left(\left\lceil \frac{d_{ij}-J_{ij}}{T_i} \right\rceil - 1\right)_0$; $sum \leftarrow sum + nC^h_{ij}$;
4: if $d_{ij} + d^h_{ij} \le D$ then
5:     $x_i \leftarrow \left\lfloor \frac{J_{ij}+\varphi_{ijk}}{T_i} \right\rfloor$;
6:     $sum \leftarrow sum + x_i C^h_{ij}$; $n_{inst} = \left\lceil \frac{t-\varphi_{ijk}}{T_i} \right\rceil$;
7:     for $1 \le p \le n_{inst}$ do
8:         $De = \varphi_{ijk} + (p-1)T_i + d_{ij} + d^h_{ij}$;
9:         if $De \le D$ then $sum \leftarrow sum + C^h_{ij}$;
10: return sum;
```

Similarly, Equation (7.7) needs to be modified to:

$$L = \sum_{\forall i,j} \left\lceil \frac{L+J_{ij}}{T_i} \right\rceil (C_i + m_i s + s + C^h_{ij}) \tag{7.35}$$

Essentially extending the execution time of a job by C^h_{ij} because, in the worst-case, the exception handler is triggered at the last instant of time in the execution of the job, thus, placing a demand on the processor equal to the total time of the job and the handler. The rest of the analysis remains unchanged.

7.6 Dynamic Jitter and offsets

In this section we briefly indicate how the iterative techniques first developed by Palencia and Harbour in [70], based on Tindell and Clark's Holistic analysis [89], and later improved in [38,69,72] can be used to provide response time analysis of distributed systems programmed using STM. From the analysis in Sections 7.4.2 and 7.5.1, we can perform response time analysis of systems where concurrency control is managed using STM, tasks have offsets and jitters, and failures are possible. Initially, the offset of each task is set to the minimum possible completion time of its predecessor, i.e.,

$$\phi_{ij} = \sum_{i \leq k \leq j} (\delta_{ik} + C_{ij}) + \delta_{ij} \quad \forall 1 \leq j \leq N_i \tag{7.36}$$

where δ_{ij} is the communication delay between nodes i and j. The jitters are all set to zero and the response time, R_{ij}, is computed using either the analysis in Section 7.4.2, if failures are not being considered, or Section 7.5.1, if we wish to consider failures. Then, jitters are modified as follows:

$$J_{i1} = 0 \tag{7.37}$$
$$J_{ij} = R_{ij-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \leq N_i \tag{7.38}$$

Offsets remain unchanged. Essentially, this means that the jitters are modified so that each task, τ_{ij}, is released at most δ_{ij}, the communication delay, time units after the completion of its predecessor τ_{ij-1}. We then compute the response times again using either the analysis in Section 7.4.2 or 7.5.1. This process is repeated until the result of two successive iterations are the same, at which point we have obtained the response time for each task. If the response times do not diverge, the process above is guaranteed to converge to the solution since the process is monotonic in its parameters. Naturally, during the computation only the contribution of the tasks running on the same processor is taken into account when computing the response times.

7.7 Experiments

In this section, we experimentally evaluate the performance of the proposed algorithm against a system simulated using RTNS [68]. In the first set of experiments, we measure the average ratio, R_{ana}/R_{sim}, between the response time of the analysis to the response time of the simulation. Execution times and periods are randomly generated as is the number of STM transactions in each task. Figure 7.2 shows the result of our first set of experiments.

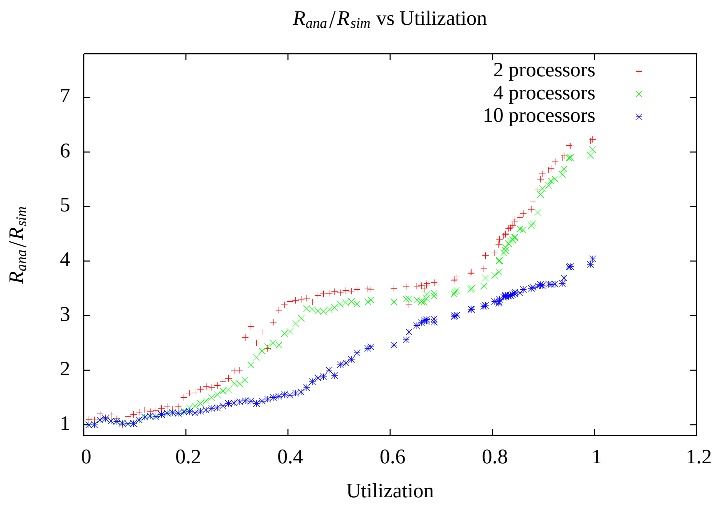

Figure 7.2: Uniprocessor: Ratio vs. Utilization

The utilization depicted on the x-axis is derived from $\sum_{\forall i} \frac{C_i + m_i s + s}{T_i} \leq 1$ [4]. Therefore, ideally, all tasks should meet their deadline for all utilizations at or below one. In this experiment, we studied three different systems. In the first system, only two processors exist and the tasks make remote invocations to either one at random. The utilization, in all three case we studied, is defined as the *maximum* utilization experienced by any node in the system. The other systems have four and ten processors respectively.

In all our experiments, the response time derived from the proposed analysis is higher than that of the simulation. This can be seen from the fact that the ratio R_{ana}/R_{sim} never falls below one in Figure 7.2. Also, the ratio becomes worse as system load increases. This occurs because as the system gets more loaded, the number of interruptions increases and hence the pessimism of the analysis increases (since we assume each interruption will result in a retry). Also, as the number of processors in the system increases, the ratio becomes better. This occurs because the utilization measured on the x-axis is the *maximum* utilization experienced by any node. Therefore, it is possible for other nodes in the system to be lightly loaded, leading to less interferences and, thus, less pessimism in the analysis.

Also, the pessimism of the analysis depends on the cost of the transactional part of the code, s, relative to the execution time of the non-transactional part of the code C_{ij}. The larger the ratio of s to C_{ij}, the more pessimistic the analysis becomes, because the pessimism in the proposed analysis is in the number of retries. Increasing the weight of the retries in the analysis by increasing the cost of the transactional component of the code results in larger estimates of worst-case response times. Figure 7.3 shows the result of an experiment where we compare the performance of a system to other systems that have a value of s twice as large and half as large as the base system.

It can be seen that the larger values of s cause greater divergence from the simulation results.

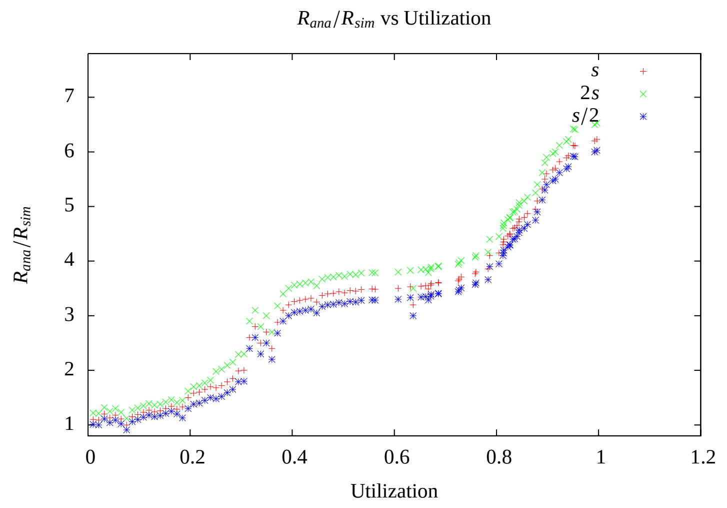

Figure 7.3: Uniprocessor: Ratio vs. Utilization

7.8 Conclusion

In this chapter we presented an algorithm for computing an upper bound on the response time of tasks in a distributed real-time system where concurrency control is managed using STM and nodes are subject to crash failures. We compared the result of our analysis to a simulation of the system in order to determine the efficacy of the proposed solution.

The result of this study indicates that it is possible to provide timeliness assurances for distributed systems programmed using STM. This allows for the first time, the usage of STM as a concurrency control mechanism (among others) for programming distributed real-time systems. Future research includes studying the different approaches for incorporating STM as outlined in [31] and dealing with some of its shortcomings. For example, we can consider whether buffered I/O can be used in STM code blocks and whether it is possible to eliminate the problem of weak atomicity using declarative languages or placing restrictions on the use of variables in imperative programming languages. Other areas of research include reducing the overhead of STM using software-hardware hybrid techniques and implementation optimizations. We also plan to consider aperiodic tasks and overload scheduling (i.e., providing assurances during overload conditions).

Chapter 8

Response time analysis of multi-processor distributed systems using STM

8.1 Introduction

In this chapter, we present an algorithm for computing a worst-case bound on the response time of tasks in a real-time distributed multiprocessor system (we define a distributed multiprocessor system as a distributed system where each node is a multiprocessor), where failures may occur and concurrency control is managed using STM.

We consider using STM as the concurrency control mechanism for (non-I/O code in) distributed real-time systems. Toward this, we propose a method for computing an upper bound on the worst-case response time of periodic tasks, programmed using STM, running in a distributed real-time system that employs PFair [6].

8.2 Roadmap

In this chapter, we provide timeliness assurances for multiprocessor distributed systems programmed using STM. Pfair scheduling is an optimal scheduling algorithm for multiprocessor real-time systems during underload conditions [6]. Therefore, we propose an algorithm for computing an upper bound on the worst-case response times of tasks running on distributed multiprocessor systems scheduled using the Pfair discipline. Towards that end, we first show how Pfair scheduling on a single processor can be represented as EDF scheduling of the transactional model proposed in [69]. We then show how an upper bound can be placed on Pfair scheduled systems on a multiprocessor and extend this result using holistic analysis to deal with distributed systems. We then show how to incorporate the retry overhead of STM into the analysis. Finally, we show how our analysis can be extended to deal with crash failures.

8.3 Pfair scheduling on a single processor

We consider periodic tasks scheduled using the Pfair discipline on a single processor. Each task, τ_i, is characterized by its period, T_i, and its execution time C_i. We assume that the deadline of each task is equal to its period. The idea of Pfair scheduling [6] is to divide the processor time among the tasks in proportion to their rates (defined as $wt(\tau_i) = C_i/T_i$). To do this, a task, τ_i, is subdivided into several quanta sized subtasks, τ_{ij}, pseudo-release times, $r(\tau_{ij})$, and pseudo-deadlines, $d(\tau_{ij})$, are derived for these subtasks and then they are scheduled using the EDF discipline (ties are broken using tie breaking rules [6]).

In the rest of the chapter, we refer to pseudo-deadlines and pseudo-release times as simply deadlines and release times for simplicity. For synchronous tasks, the deadlines and release times of subtasks can be derived using the following equations:

$$r(\tau_{ij}) = \left\lfloor \frac{j-1}{wt(\tau_i)} \right\rfloor \quad (8.1)\ , \quad d(\tau_{ij}) = \left\lceil \frac{j}{wt(\tau_i)} \right\rceil \quad (8.2)$$

For asynchronous tasks, assume that task τ_i releases its first subtask at time r and let τ_{ij} $(j \geq 1)$ be this task. The release time and deadline of each subtask τ_{ik} $(k \geq j)$ can be obtained by computing the term $\Delta(\tau_i) = r - \lfloor (j-1)/wt(\tau_i) \rfloor$ and adding it to Equations (8.1) and (8.2).

In [69], the authors show how it is possible to perform schedulability analysis for EDF systems programmed using "transactions". A transaction, as used in [69], is a sequence of tasks that belong to a single programming context. A sort of precedence constraint is placed on the relative execution times of these tasks by using offsets and jitters. It can be easily shown that this transactional model can be used to represent tasks scheduled using the Pfair discipline.

Specifically, since each task in Pfair scheduling is subdivided into subtasks and these subtasks belong to the same execution context, we can represent each task as a transaction. It now becomes necessary to obtain values for the jitter and offsets to specify the precedence constraints of the subtasks (i.e, τ_{ij} can only start executing after τ_{ij-1} has completed executing). In Section 8.3.1 we show how this is performed.

8.3.1 Application to Pfair scheduling

In this Section, we show how the analysis of [69] can be applied to Pfair scheduled systems. As mentioned before, in Pfair scheduling, each task, τ_i, is subdivided into several quantum length subtasks, τ_{ij}. Therefore, we first need to determine the scheduling parameters of these subtasks. The execution time of each subtask is one quantum (i.e., $C_{ij} = Q$, where Q is the duration of a scheduling quantum). To keep the analysis simple, we shall assume $Q = 1$, it is trivial to extend this analysis for $Q \geq 1$. Each subtask retains the period of its parent task, T_i, and d_{ij} is set to the value of Equation (8.2). We now turn our attention to the offset, ϕ_{ij}, and jitter, J_{ij} of our subtasks.

The initial values of the offsets, ϕ_{ij} are set as follows:

$$\phi_{ij} = \begin{cases} \sum_{\forall k<j} C_{ik} & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{8.3}$$

and all jitters are set to the pseudo-release times of each subtask as computed in Equation (8.1), i.e., $J_{ij} = r(\tau_{ij})$. We then perform the analysis in [69] and update jitters, after the analysis, as follows:

$$J_{ij} = \begin{cases} R_{ij-1} - \phi_{ij} & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{8.4}$$

where R_{ij} is the response time of τ_{ij}. This process is repeated until two successive iterations produce the same response time, at which point we have the worst-case response time of the tasks in the system.

8.4 Multi-processors

Now that we have established that Pfair scheduling can be represented as EDF scheduling using a transactional model, we turn our attention to providing an upper bound on the worst-case response time of Pfair scheduled multi-processor systems. The following theorem shows how this can be done

Theorem 55 (Theorem 6 in [7]). *An upper bound on the response time of a task τ_k in an EDF-scheduled multiprocessor system can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{ub} = C_k$:*

$$R_k^{ub} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} I_k^i(R_k^{ub}) \right\rfloor \tag{8.5}$$

with $I_k^i(R_k^{ub}) = \min(W_i(R_k^{ub}), J_k^i(D_k), R_k^{ub} - C_k + 1)$

The proof of Theorem 55 can be found in [7]. The term $W_i(R_k^{ub})$ is the maximum workload offered by τ_i during a period of duration R_k^{ub}, $J_k^i(D_k)$ is the maximum number of interferences by τ_i that can occur before the deadline of τ_k, and $R_k^{ub} - C_k + 1$ is a natural upper bound on the interference of any task on τ_k (because the response time, R_k^{ub}, is naturally composed of a period of time during which τ_k executes, C_k, and some interferences $R_k^{ub} - C_k + 1$). In order to take advantage of Equation (8.5), we need to derive expressions for these terms using our transactional model.

First, let us consider the term $W_i(R_k^{up})$. Before presenting the analysis we change the notation to $W_i(R_{ab}^{up})$, since we will be analyzing τ_{ab}. The maximum workload offered by τ_i during a period of length R_{ab}^{up} is equal to the maximum number of jobs of τ_i that can execute during that period . Remember, however, that, in Pfair scheduling, each task, τ_i, is divided into several subtasks to create a transaction Γ_i. So, in essence, when computing the term $W_i(R_{ab}^{up})$, we are computing the worst-case contribution of transaction Γ_i. From the analysis in [69], we know that the worst-case contribution of Γ_i to the response time of a task being analyzed during a period of length R_{ab}^{up} occurs when one of its tasks τ_{ik} coincides with the start of the period. We also know that when τ_{ik} coincides with the beginning of the period, the worst-case contribution of a task τ_{ij} during a period of length R_{ab}^{up} is:

$$W_{ijk}(R_{ab}^{up}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{R_{ab}^{up} - \varphi_{ijk}}{T_i} \right\rceil \right)_0 C_{ij} \tag{8.6}$$

where $\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) \mod T_i$. Thus, the worst-case contribution, workload, of a transaction Γ_i to the response time of a task τ_{ab} when τ_{ik} coincides with the beginning of the period is:

$$W_{ik}(R_{ab}^{up}) = \sum W_{ijk}(R_{ab}^{up}), \quad \forall j \in \Gamma_i \tag{8.7}$$

and the upper bound on the contribution of Γ_i is:

$$W_i^*(R_{ab}^{up}) = \max\left(W_{ik}(R_{ab}^{up})\right), \quad \forall k \in \Gamma_i \tag{8.8}$$

Likewise, we can determine a value for $J_k^i(D_k)$ from the analysis in [69]. Specifically:

$$J_{ijk}(D_{ab}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lfloor \frac{D_{ab} - \varphi_{ijk} - d_{ij}}{T_i} \right\rfloor + 1 \right)_0 C_{ij} \tag{8.9}$$

$$J_{ik}(D_{ab}) = \sum J_{ijk}(D_{ab}), \quad \forall j \in \Gamma_i \tag{8.10}$$

$$J_k^i(D_k) = J_{ab}^i(D_{ab}) = J_i^*(D_{ab}) = \max\left(J_{ik}(D_{ab})\right), \quad \forall k \in \Gamma_i \tag{8.11}$$

Unlike in [69], we do not have to take into account the interference of tasks belonging to the transaction being analyzed for two reasons. First, by definition of Pfair scheduling, our subtasks have precedence constraints. Thus, when analyzing a subtask we are sure that all previous subtasks have finished executing and all subsequent subtasks are yet to start. Second, since we assume that deadlines are equal to periods for transactions (see Section 8.3), we are sure that while analyzing a particular transaction its predecessor's deadline has already passed (and so it is no longer in the system) and its successor is yet to start (otherwise the period would have been over and the current transaction's deadline would have already passed).

Finally, we need to set the value of the offset and jitter for the subtasks in our analysis. We take a pessimistic approach and set the jitter of each subtask to one quanta after the deadline of its preceding subtask i.e.,

$$J_{ij} = \begin{cases} d_{ij-1} + 1 & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{8.12}$$

and the offset of each subtask is set to the best case completion time of its predecessor as in Equation (8.3).

8.5 Distributed Multiprocessor Systems

Now that we have shown, in Section 8.4, how to obtain an upper bound on the response time of tasks scheduled using the Pfair discipline on a multiprocessor, we can extend our analysis to a distributed system. Specifically, we can use the variant of holistic analysis developed in [69], i.e., task offsets are initially set to the best-case completion time of their predecessor:

$$\phi_{ij} = \sum_{i \le k \le j} (\delta_{ik} + C_{ij}) + \delta_{ij} \quad \forall 1 \le j \le N_i \tag{8.13}$$

where δ_{ij} is the communication delay between nodes i and j. The jitters are all set to zero and the response time, R_{ij}, is computed using the analysis in Section 8.4. Then, jitters are modified as follows:

$$J_{i1} = 0 \tag{8.14}$$

$$J_{ij} = R_{ij-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \le N_i \tag{8.15}$$

Offsets remain unchanged. Essentially, this means that the jitters are modified so that each task, τ_{ij}, is released at most δ_{ij}, the communication delay, time units after the completion of its predecessor τ_{ij-1}. We then compute the response times again using the analysis in Section 8.4. This process is repeated until the result of two successive iterations are the same, at which point we have obtained the response time for each task. If the response times do not diverge, the process above is guaranteed to converge to the solution since the process is monotonic in its parameters. Naturally, during the computation only the contribution of the tasks running on the same processor is taken into account when computing the response times.

8.6 Considering STM

For the sake of this analysis, we consider atomic regions programmed using STM (e.g, [39]). We assume that each atomic region is kept short and that all atomic regions have computation cost at most s. We now show how the retry overhead of these atomic regions can be incorporated into our analysis. In [47], Anderson *et. al.* show how the overhead of lock-free code can be incorporated into Pfair scheduled systems.

Here, we show how this analysis can be applied to atomic regions programmed using STM. We assume that any two atomic regions that execute concurrently can interfere with each other. We further assume that a retry can only occur at the completion of an atomic region (i.e., validation of the transaction is performed before it commits). The second assumption implies that the number of retries of an atomic region is at most the number of concurrent accesses to atomic regions by other tasks. Finally, we assume that each atomic region is small and spans, at most, two quanta. The last assumption makes sense since atomic regions are usually designed to be small in order to reduce the likelihood of interferences and hence retries.

The idea behind the analysis is to compute the worst-case overhead introduced by the retry behavior of atomic regions. This overhead is then added to the execution time of each task to compute its worst-case demand on the processor. Using these new execution times, the analysis in Sections 8.4 and 8.5 can be used to compute the response time on a distributed system.

We assume that each task, τ_i, has N_i atomic regions and accesses atomic regions at most AA_i times during each quantum. Thus, the maximum number of interferences that can occur to τ_a in a single quantum is:

$$I_a = \mathbf{maxsum}_{M-1}\{AA_i | i \neq a\} \tag{8.16}$$

where M is the number of processors. Also, since we assume that an atomic region can span at most two quanta, and hence can only be preempted once, the overhead introduced by a single access to an atomic region in τ_a can be computed as in Equation (8.17).

$O_a^{one} = s + (2I_a + 1)s \quad (8.17)\ , \quad O_a = O_a^{one} \times N_a \quad (8.18)$

The term $(2I_a + 1)s$ in Equation (8.17), represents the overhead of retries. The $2I_a$ represents the maximum number of retries that may occur in the two quanta that the operation spans, and the 1 added to $2I_a$ represents the retry that may occur due to interference that occurred while the task was preempted in the middle of its atomic region (note that by our assumption this can only occur once). Now that we have computed the overhead of one atomic region, we can compute the overhead of the N_a atomic regions using Equation (8.18).

Thus, we can set the new execution time of task τ_a to $C_a = O_a + C_a$ and then perform the analysis of Sections 8.4 and 8.5 using this new value.

8.7 Handling Failures

In this section we show how the analysis in Section 8.4 can be extended to take failures into account. We assume that each quantum-sized subtask in the system has an associated exception handler that can be used to restore the system to a safe state in case of failure, and that this exception handler has execution time C^h_{ji} and relative deadline d^h_{ij}. The absolute deadline of the handler is relative to the time failure occurs, t_f, thus the absolute deadline of the handler is $D^h_{ij} = t_f + d^h_{ij}$. Since we cannot determine t_f a priori, we assume a worst-case scenario where each job executes to completion, using up as much processor time as possible, and then an error occurs that triggers its exception handler. It is this worst-case scenario from which we derive the deadline of the exception handlers.

8.7.1 Analysis

In order to incorporate exception handlers in the analysis, it is necessary to extend Equations (8.6) and (8.9) to take their overhead into account.

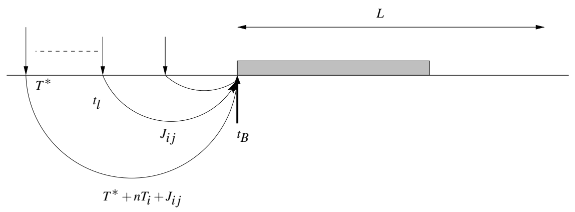

Figure 8.1: Scenario for calculating worst-case contribution

We assume that a critical instant occurs at time t_B (note that for this analysis we do not need to know the value of t_B, we just assume that it exists) and compute the worst-case contribution of the exception handlers during a period of length L starting at t_B. From [69], we know that there are

$$n_1 = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{8.19}$$

jobs of τ_{ij} that can start at the beginning of the period being studied after suffering some jitter. Each of these activations has an associated exception handler that can start, at most, at time $t_B + d_{ij}$. Thus, when computing the maximum number of interferences that can occur during a period of length L, we only consider these exception handlers if $d_{ij} < L$. There are

$$n_2 = \left\lceil \frac{L - \varphi_{ijk}}{T_i} \right\rceil \tag{8.20}$$

activations that will occur within a period of duration L. The latest start time of the exception handlers of these activations are

$$S = \varphi_{ijk} + (p-1)T_i + d_{ij} \qquad \forall p = 1 \cdots n_2 \tag{8.21}$$

We only consider exception handlers for which $S < L$. We also need to compute the overhead of exception handlers whose activations do not contribute to the overhead. This may occur when an activation finishes before the critical instant t_B, but its exception handlers execute after t_B. In Figure 8.1, the first activation that can be delayed to start at the beginning of the period being studied is depicted as starting at t_l.

However, if failures are not considered, jobs that start before t_l, such as the one depicted as starting at time T^* in Figure 8.1, do not contribute to the analysis because even if these jobs were delayed J_{ij}, their activation time would fall before t_B. It is, now, possible for these tasks to contribute to the worst-case response time if their exception handlers start after t_B. The latest start time of a handler whose job arrives at T^* is:

$$S = T^* + d_{ij} \tag{8.22}$$

If S is greater than or equal to t_B then the handler will contribute to the busy period. In other words:

$$T^* + d_{ij} \geq T^* + nT_i + J_{ij} \tag{8.23}$$

which gives us:

$$n < \frac{d_{ij} - J_{ij}}{T_i} \tag{8.24}$$

Since n is an integer, Equation (8.24) resolves to:

$$n = \left(\left\lceil \frac{d_{ij} - J_{ij}}{T_i} \right\rceil - 1 \right)_0 \tag{8.25}$$

The zero subscript indicates that negative values are considered zero (in this case, such an event indicates that none of the jobs starting before t_l can contribute to the busy period). Thus we can compute the contribution of the exception handlers of τ_{ij} to the response time of τ_{ab} in a period of duration L, when the task whose activation time coincides with t_B is τ_{ik} using Algorithm 18. Thus, Equation (8.6) becomes:

$$W_{ijk}(R_{ab}^{up}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{R_{ab}^{up} - \varphi_{ijk}}{T_i} \right\rceil \right)_0 C_{ij} + W_{ijk}^{h}(R_{ab}^{up}) \tag{8.26}$$

and Equation (8.9) becomes:

$$\begin{aligned} J_{ijk}(D_{ab}) = & \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lfloor \frac{D_{ab} - \varphi_{ijk} - d_{ij}}{T_i} \right\rfloor + 1 \right)_0 C_{ij} \\ & + W_{ijk}^{h}(D_{ab} - d_{ij}^{h}) \end{aligned} \tag{8.27}$$

Algorithm 18: $W_{ijk}^{h}(L)$

```
1: $sum = \left( \left\lceil \frac{d_{ij} - J_{ij}}{T_i} \right\rceil - 1 \right)_0 C_{ij}^{h}$;
2: if $d_{ij} < L$ then
3:     $sum \leftarrow sum + \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor C_{ij}^{h}$;
4: $n_2 \leftarrow \left\lceil \frac{L - \varphi_{ijk}}{T_i} \right\rceil$;
5: for $1 \le p \le n_2$ do
6:     $S \leftarrow \varphi_{ijk} + (p-1)T_i + d_{ij}$;
7:     if $S < L$ then
8:         $sum \leftarrow sum + C_{ij}^{h}$;
9: return sum;
```

The rest of the analysis remains unchanged.

8.8 Experiments

In this section, we perform a number of experiments to verify the validity of the analysis presented. In our first set of experiments, we determine whether or not the analysis presented in Section 8.4 can be used to derive suitable upper bounds for the response times of Pfair scheduled tasks. Toward that goal, we conducted a number of experiments to determine the Deadline Satisfaction Ratio (or DSR) of the analysis in Section 8.4.

We perform the analysis for ten tasks on multiprocessors that contain 4, 6 and 8 processors. For each of these systems we fix the execution time of the tasks and vary the periods to obtain utilizations between 0 and m, where m is the number of processors. We ran our experiment 700 times and recorded the average DSR for each utilization. Figure 8.2 depicts the result of the experiments. The utilization on the x-axis is normalized with respect to the number of processors used.

The results indicate that the analysis is tighter for a smaller number of processors, but that it provides a good bound for response time in most cases (for example, the average DSR in our experiments does not drop below 0.8 until close to the 0.8, normalized, system utilization point). In the next set of experiments, we compute the ratio of the response time obtained using the proposed analysis to a system simulated using [68].

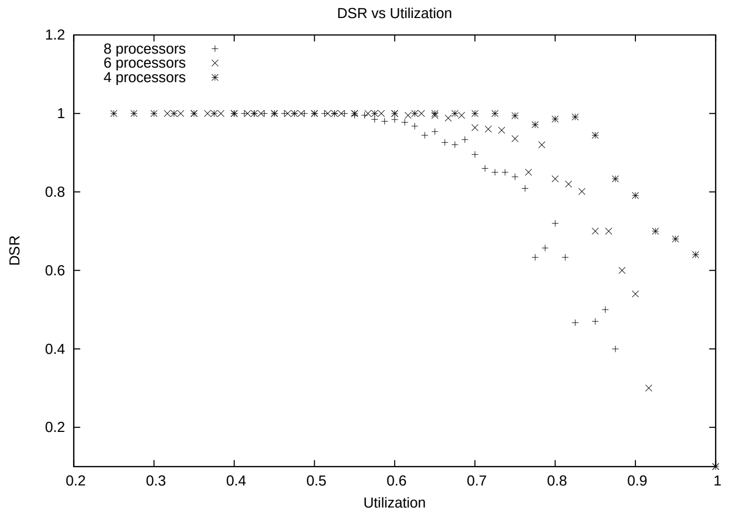

Figure 8.2: DSR vs. Utilization

For this experiment, we fixed the number of processors and nodes, fixed the execution times and varied the periods to obtain utilizations between 0 and m. Figure 8.3 depicts the result of our experiments. As can be seen, the response time analysis becomes more pessimistic as the value of s increases due to the dependence of the analysis on Equation (8.18). Other sources of pessimisim include Equations (8.5) and (8.12).

8.9 Conclusion

We presented an algorithm for computing an upper bound on the worst-case response time for tasks on a multiprocessor distributed real-time system where concurrency control is programmed using STM.

With this result, it is now possible to include STM in the repertoire of real-time programming tools on such architectures. Future work includes tightening the analysis by considering slack (as in [7]), and considering non-periodic tasks and overload scheduling. Other directions include investigating other methods [31] for incorporating STM in distributed real-time

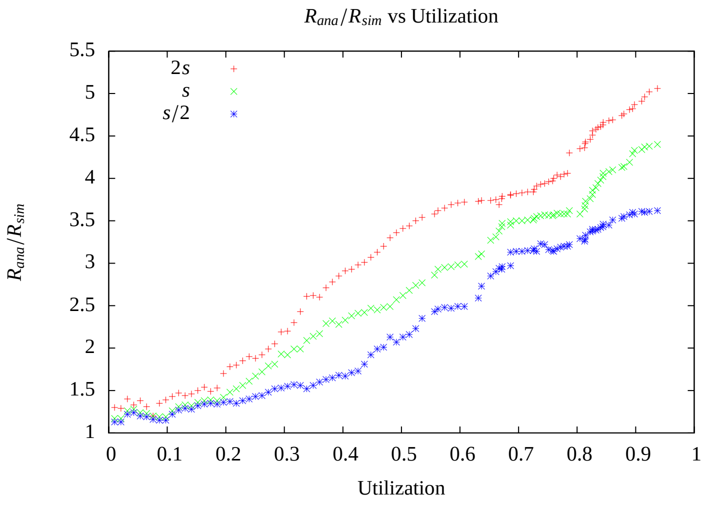

Figure 8.3: R_{ana}/R_{sim} vs. Utilization

systems.

Chapter 9

Conclusions, Contributions and Proposed Post Preliminary Exam Work

In this proposal, we addressed the problem of scheduling distributable threads in distributed real-time systems. Specifically, we investigated the possibility of providing timeliness assurances to distributed real-time systems that are non-synchronous and that are subject to failure.

Our research clarified a number of issues. First, it indicates that collaborative scheduling algorithms can provide better timeliness assurances than independent node scheduling algorithms. This improved timeliness is achieved because all nodes in the system have full information of what is going on and therefore can make decisions that optimize system-wide timeliness. In contrast, the lack of global information at each node in independent node scheduling forces these nodes to make scheduling decisions using local information only, thus, possibly, jeopardizing global timeliness.

However, the improved timeliness of collaborative scheduling comes at the price of higher scheduling overhead and thus can benefit only those systems that can tolerate their higher overhead. We also believe that collaborative scheduling, since it incorporates failure detection with scheduling, allows us to more seamlessly provide performance assurances during failures.

The high overhead of collaborative scheduling becomes especially obvious when we consider distributed dependencies. We showed in this proposal that the overhead introduced by distributed lock management and distributed deadlock detection and resolution algorithms are quite significant.

Thus, we proposed an alternative concurrency control mechanism. Specifically, we addressed the use of software transactional memory for concurrency control in distributed real-time systems. We proposed two different schedulability analysis algorithms for distributed systems programmed using software transactional memory for concurrency control. The schedulability analysis we proposed allows programmers of distributed real-time systems to add software transactional memory to their repertoire of tools.

9.1 Contributions

In this proposal, we studied a number of issues. In particular, our research on collaborative scheduling resulted in the design of three different collaborative scheduling algorithms. The first of these, ACUA, is based on the distributed consensus problem. ACUA is designed to run on a partially synchronous distributed system where both message loss and communication delay are stochastically described.

Thus, our first step in designing ACUA was to determine the feasibility of implementing one of the Chandra-Toueg failure detectors on such a platform [13]. In Chapter 3, we showed how a stochastic $\mathcal{S}$-class failure detector can be designed in such an environment and how this FD can be used to implement a solution to the distributed consensus problem.

We then proposed a collaborative scheduling algorithm that uses this solution to the distributed consensus problem to come to agreement on scheduling decisions. We empirically and analytically evaluated the properties of ACUA and showed that it could provide better timeliness assurances than independent node scheduling for systems that could tolerate its higher overhead.

In an attempt to lower the overhead associated with the consensus algorithm (particulary in the presence of failure), we designed a quorum-based collaborative scheduling solution, QBUA. QBUA operates in the same environment as ACUA but does not depend on a solution to the distributed consensus problem. In Chapter 4, we described this algorithm and empirically and analytically evaluated its properties. The same was performed for an extension of QBUA, DQBUA, that allowed the algorithm to handle distributed dependencies (described in Chapter 5).

At this point in our research, we identified distributed concurrency control as one of the major sources of overhead in these algorithms. Thus we turned our attention to alternatives to lock-based concurrency control solutions. In particular, we identified software transactional memory as a promising solution to this problem.

Toward that end, we designed two schedulability analysis algorithms for providing upper bounds on the response times of tasks in distributed systems programmed using STM (Chapters 7 and 8). The first of these algorithms provides such assurances for distributed systems where nodes are uniprocessors while the second is an algorithm that targets distributed systems where each node is a multiprocessor. To further our goal of making STM a part of distributed real-time systems, we identified some of the issues that need to be resolved in order to do so. Some of these issues and their proposed solutions are discussed in Chapters 1 and 6. In Section 9.2, we summarize some of these issues which we propose to address after the preliminary exam.

9.2 Post Preliminary-Exam Work

We propose the following post preliminary-exam work.

- **Investigate Different Progress Guarantees for Real-Time STM**: As mentioned in Chapter 1, we intend to study the different progress guarantees that can be provided using STM. STM systems are usually designed to be obstruction-free and progress guarantees are provided by an out of bounds contention manager.

  An obstruction-free algorithm is different from algorithms that provide stronger non-blocking progress guarantees (like wait-freedom and lock-freedom, as explained in Chapter 1), in that it separates the notion of correctness from the notion of progress. Essentially, an obstruction-free algorithm only guarantees progress in the absence of contention. However, it always maintains its algorithmic invariants — in our case, the invariant is the semantics of STM.

  Thus, requiring STM implementations to be obstruction-free is a good design choice since it allows vendors to concentrate on implementing correct algorithms. The vendor, end-user, or third parties can then provide orthogonal contention managers to provide the progress guarantees needed for a particular application.

  The function of a contention manager is to determine what happens when two or more STM transactions need to access the same block of memory. When such a scenario occurs, some scheduling discipline is required to determine which transaction gets aborted. There have been a number of different contention management policies proposed, but none of these have considered real-time constraints. The following is a brief list of some of the most common contention management policies considered in the literature:

  - **Aggressive:** This contention manager always chooses to abort an enemy (or conflicting) transaction at conflict time. This is the most basic contention manager and is the one we consider in our current response time analysis in Chapters 7 and 8 (since we assume that an abort occurs whenever any two conflicting transactions execute).
  - **Polite:** This contention manager uses exponential back-off techniques, similar to that used in Ethernet, to determine which transaction to abort.
  - **Randomized:** This contention manager randomly chooses which transaction to abort.

  - **Karma:** The Karma contention manager aborts the transaction with the least amount of work performed when a conflict occurs.
  - **Eruption:** This contention manager increases the priority of a transaction that others are waiting on in order to make it complete faster (priority is used in arbitrating which transaction is aborted). The idea of this contention manager is similar to the popular priority inheritance technique used in lock-based real-time concurrency control.
  - **Kindergarten:** This policy encourages transactions to take turns accessing shared memory (like children accessing toys or other shared objects in its namesake).
  - **Timestamp:** This policy attempts to be as fair as possible, and uses timestamps to arbitrate among the transactions trying to access a shared memory block.
  - **QueueOnBlock:** This policy causes conflicting transactions to wait in a queue for the transaction currently holding the shared block and spin on a "finished" flag that is set by the enemy transaction at its commit time. If the transaction has waited for too long, the definition of too long is application dependent, the transaction aborts the enemy transaction and acquires the shared block.

  Other contention management policies exist, but these are some of the most commonly studied policies. As can be seen, none of them consider real-time constraints. We intend to consider real-time parameters for contention management (such as deadlines or potential utility density) and see how these contention managers affect the timeliness assurances we can provide.

- **Implementing Cache Coherence Protocols for STM**: This approach is discussed in detail in Chapter 6, but we reiterate some of the most important points about it here. Implementing STM on top of a distributed cache coherence protocol has been investigated in [43, 61]. In this approach, code is immobile, but data objects move among nodes as required. The approach uses a distributed cache coherence protocol to find and move objects. We intend to design real-time cache coherence protocols, where timeliness is an integral part of the algorithm. We plan to design STM on top of these protocols and compare their performance to the flow control abstraction. An important advantage of this approach is that it eliminates the need for a distributed commit protocol. Since distributed commit protocols are a major source of inefficiency in real-time systems [35], such an approach is expected to yield better performance.

  The cache coherence protocol needs to be location aware in order to reduce communication latency and should be designed to reduce network congestion. The cache coherence problem for multiprocessors has been extensively studied in the literature [83]. There are also some solutions for the distributed cache coherence problem (see [1,15,52,87] for a, not necessarily representative, sample of research on this issue). Distributed cache coherence bears some similarity to distributed hash table (or DHT) protocols which have been an active topic of research recently due to the popularity of peer-to-peer applications. Examples of DHT algorithms that are of interest are [45, 73, 77].

We envision a cache coherence algorithm based on hierarchical clustering to reduce network traffic and path reversal to synchronize concurrent requests, an approach used in [43]. Other approaches for implementing distributed cache coherence will also be considered. An important part of our research in this area will be to design cache coherence protocols that can provide timeliness guarantees that we can verify theoretically and empirically.

- **Compiler instrumentation for STM:** As mentioned in Chapter 1, one of the major hurdles to mainstream acceptance of STM is the overhead associated with the unnecessary instrumentation of loads and store.

  If only loads and stores within transactions are instrumented, this may lead to weak atomic semantics where non-transactional access to shared memory can lead to violation of atomicity properties. In addition, a careful analysis of the code in an application can reduce the number of loads and stores that need to be instrumented and hence reduce the overhead of STM.

  Thus, what is required is some sort of static analysis at compile time that would allow us to minimize the number of loads and stores to instrument while at the same time, possibly, preventing weak atomicity by ensuring that all loads and stores that need instrumenting are in fact instrumented. This problem is akin to the problems of alias analysis and escape analysis in standard compiler theory.

  From lessons learned from research on alias analysis and escape analysis, we know that this problem is quite challenging. However, we believe that most real-time code, in contrast to general purpose code, is more structured and would allow a reasonably efficient implementation of such an analysis.

  We intend to develop algorithms to automatically instrument loads and stores where necessary and test the efficacy of these algorithms both analytically and empirically.

- **Integrating hard and soft real-time analysis for STM**: Our current schedulability analysis (see Chapters 7 and 8) for systems where concurrency control is managed using STM has concentrated on traditional hard real-time systems. While this analysis provides a good idea about the schedulability of a system, it does not say anything about the type of assurances that can be offered during overloads.

  As mentioned in Chapter 1, there are a number of emerging distributed real-time applications that operate in environments where transient or sustained overloads are possible. Thus, it becomes necessary to extend our analysis to include these cases.

  Time Utility Functions, as previously stated, are ideally suited for describing the timeliness requirements in such systems since they provide us with the ability to describe an activity's urgency and importance separately. Utility Accrual schedulers allow us to take advantage of the descriptive power of TUFs and provide timeliness assurances that gracefully degrade during overloads.

We intend to extend our current schedulability analysis to enable them to provide timeliness assurances for systems scheduled using Utility Accrual schedulers. This will increase the number of systems for which our results can be applied.

- **Hybrid data/code migration**: We touched on the research issues involved in developing a hybrid data/code migration scenario for reducing the overhead of STM in Chapter 6. Here we reiterate some of the most salient points for this direction of research. This approach is touched upon in [10]. This is a hybrid approach where either data objects or code can migrate while still retaining the semantics of STM. By allowing either code or data to migrate, we can choose a migration scenario that results in the least amount of communication overhead. For example, suppose we have a simple transactional program that increments the value of a shared variable X and stores the new value in the transactional store. Assume further that X is remote, using a data flow abstraction would necessitate two communication delays; one to fetch X from its remote location and the other to send it back once it has been incremented. Using a control flow abstraction in this case may be more efficient since it will only involve a single communication delay.

  On the other hand, assume that several processes need access to a small data structure and that these processes are in roughly the same location and are far away from the data they need. Since communication delay depends on distances, it may make sense to migrate the data to the processes in this case rather than incur several long communication delays by moving the code to the data. In short, the choice of whether to migrate code or data can have a significant effect on performance. In [10], this is accomplished under programmer control by allowing an *on* construct which a programmer can use to demarcate code that should be migrated. We intend to elaborate on this by coming up with solutions that would use static analysis at compile-time (or dynamically at run-time) to make decisions about which part of the application to move using a number of heuristics such as, for example, size of code/data and locality considerations.

Bibliography

[1] J. Aguilar and E. L. Leiss. A general adaptive cache coherency-replacement scheme for distributed systems. In *IICS '01: Proceedings of the International Workshop on Innovative Internet Computing Systems*, pages 116–125, London, UK, 2001. Springer-Verlag.

[2] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC '02*, pages 354–370. Springer-Verlag, 2002.

[3] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *IEEE RTSS*, pages 92–105, 1996.

[4] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *IEEE RTSS*, pages 28–37, 1995.

[5] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*. Amsterdam: Kluwer Academic Publishers., 1997.

[6] J. H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68(1):157–204, 2004.

[7] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *IEEE RTSS*, pages 149–160, 2007.

[8] R. Bettati and J. W. s. Liu. End-to-end scheduling to meet deadlines in distributed systems. pages 452–459, 1992.

[9] J. Bobba, R. Rajwar, and M. Hill. Transactional memory biblography. `http://www.cs.wisc.edu/trans-memory/biblio/swtm.html`.

[10] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, 2008.

[11] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 285–301, New York, NY, USA, 1984. ACM.

[12] J. R. Cares. *Distributed Networked Operations: The Foundations of Network Centric Warfare*. iUniverse, Inc., 2006.

[13] T. D. Chandra. Unreliable failure detectors for asynchronous distributed systems. Technical Report TR93-1377, 1993.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[15] Y. Chang and L. N. Bhuyan. An efficient tree cache coherence protocol for distributed shared memory multiprocessors. *IEEE Transactions on Computers*, 48(3):352–360, 1999.

[16] W. Chen, S. Lin, Q. Lian, and Z. Zhang. Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. In *PRDC '05*, pages 7–14, Washington, DC, USA, 2005. IEEE Computer Society.

[17] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

[18] H. Cho, B. Ravindran, and E. D. Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373–384, 2007.

[19] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Trans. Softw. Eng.*, 15(1):10–17, 1989.

[20] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, 1993.

[21] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[22] E. Curley, J. S. Anderson, B. Ravindran, and E. D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *IEEE SRDS*, pages 267–276, 2006.

[23] P. H. Dana. Global positioning system (gps) time dissemination for real-time applications. *Real-Time Syst.*, 12(1):9–40, 1997.

[24] J. R. G. de Mendivil, A. Demaille, J. B. Auban, and J. R. Garitagoitia. Correctness of a distributed deadlock resolution algorithm for the single request model. In *PDP '95: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, page 254, Washington, DC, USA, 1995. IEEE Computer Society.

[25] J. R. G. de Mendívil, n. Federico Fari J. R. Garitagoitia, C. F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the and model. *IEEE Trans. Parallel Distrib. Syst.*, 10(5):433–447, 1999.

[26] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01*, pages 75–80, 2001.

[27] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, 1986.

[28] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[29] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. Technical report, Virginia Tech, ECE Dept., November 2007. Available at: `http://www.real-time.ece.vt.edu/RST_TR.pdf`.

[30] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. In *International Conference on Reliable Software Technologies - Ada-Europe 2008*, June 2008. To appear, available at: `http://www.real-time.ece.vt.edu/rst08.pdf`.

[31] S. F. Fahmy, B. Ravindran, and E. D. Jensen. On scalable synchronization for distributed embedded real-time systems. In *SEUS*, 2008. `http://www.real-time.ece.vt.edu/seus08.pdf`.

[32] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling dependent distributable real-time threads in dynamic networked embedded systems. In *DIPES*, September 2008. To appear, available at: `http://www.real-time.ece.vt.edu/dipes08.pdf`.

[33] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling distributable real-time threads in the presence of crash failures and message losses. In *ACM SAC*, pages 294–301, March 2008.

[34] J. Goldberg, L. Gong, I. Greenberg, R. Clark, E. Jensen, K. Kim, and D. Wells. Adaptive fault-resistant systems, 1994.

[35] R. Gupta, J. Haritsa, K. Ramamritham, and S. Seshadri. Commit processing in distributed real-time database systems. *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 220–229, 4-6 Dec 1996.

[36] W. A. Halang and M. Wannemacher. High accuracy concurrent event processing in hard real-time systems. *Real-Time Syst.*, 12(1):77–94, 1997.

[37] K. Han, B. Ravindran, and E. D. Jensen. Exploiting slack for scheduling dependent, distributable real-time threads in mobile ad hoc networks. In *RTNS 2007*, pages 225–234, 2007.

[38] M. G. Harbour and J. C. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *IEEE RTSS*, page 200, 2003.

[39] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402. 2003.

[40] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. *icdcs*, 00:522, 2003.

[41] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, Jul 2003.

[42] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[43] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[44] J.-F. Hermant and J. Widder. Implementing reliable distributed real-time systems with the Θ-model. In *OPODIS*, pages 334–350, 2005.

[45] K. Hildrum, R. Krauthgamer, and J. Kubiatowicz. Object location in realistic networks. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 25–35, New York, NY, USA, 2004. ACM.

[46] C. Hoare. Towards a theory of parallel programming. In C. Hoare and R. Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[47] P. Holman and J. H. Anderson. Supporting lock-free synchronization in pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66(1):47–67, 2006.

[48] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems, 1985. IEEE RTSS, pages 112–122, 1985.

[49] B. Kao and H. Garcia-Molina. Subtask deadline assignment for complex distributed soft real-time tasks. Technical report, Stanford, CA, USA, 1993.

[50] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[51] B. Kao, H. Garcia-molina, and B. Adelberg. On building distributed soft real-time systems. In *In The Third Workshop on Parallel and Distributed Real-Time Systems*, pages 13–19, 1995.

[52] C. A. Kent. Cache coherence in distributed systems. *WRL Technical Report 87/4*, 1987.

[53] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[54] T. F. Knight. An architecture for mostly functional languages. In *ACM LFP*, pages 500–519, Aug 1986.

[55] B. Korenfeld and M. Medina. Transactional memory. Technical Report MIT/LCS/TM-475, University of Tel-Aviv Computer Engineering Dept., Jun 2006.

[56] N. Krivokapić, A. Kemper, and E. Gudes. Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. *The VLDB Journal*, 8(2):79–100, 1999.

[57] A. D. Kshemkalyani and M. Singhal. A one-phase algorithm to detect distributed deadlocks in replicated databases. *IEEE Trans. on Knowl. and Data Eng.*, 11(6):880–895, 1999.

[58] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[59] P. Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Trans. Comput.*, 54(9):1138–1153, 2005. Student Member-Haisang Wu and Senior Member-Binoy Ravindran and Member-E. Douglas Jensen.

[60] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[61] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. 2006.

[62] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time java. *RTSS*, 0:62–71, 2005.

[63] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[64] S. McCanne and S. Floyd. ns-2: Network Simulator. http://www.isi.edu/nsnam/ns/.

[65] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 282–284, New York, NY, USA, 1984. ACM.

[66] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *DISC '99*, pages 49–63, London, UK, 1999. Springer-Verlag.

[67] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

[68] P. Pagano, P. Batra, and G. Lipari. A framework for modeling operating system mechanisms in the simulation of network protocols for real-time distributed systems. *IPDPS*, 0:160, 2007.

[69] J. Palencia and M. G. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. *ECRTS*, 00:3, 2003.

[70] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the 19th IEEE RTSS*, pages 26–37, 1998.

[71] G. Pardo-Castellote. Omg data-distribution service: Architectural overview. *ICDCSW*, 00:200, 2003.

[72] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. *RTAS*, pages 66–75, 2005.

[73] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM.

[74] B. Ravindran, J. S. Anderson, and E. D. Jensen. On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *IFIP SEUS Workshop*, pages 67–81, 2007.

[75] B. Ravindran, E. Curley, J. S. Anderson, and E. D. Jensen. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *ISORC '07*, pages 344–353. IEEE Computer Society, 2007.

[76] M. Roesler and W. A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *IEEE Trans. Comput.*, 38(8):1212–1224, 1989.

[77] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[78] M. Saksena and S. Hong. An engineering approach to decomposing end-to-end delays on a distributed real-time system. *Parallel and Distributed Real-Time Systems, Workshop*, 0:244, 1996.

[79] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[80] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[81] C. Shih and J. A. Stankovic. Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, Amherst, MA, USA, 1990.

[82] M. Spuri. Analysis of deadline scheduled real-time systems. Technical report, In Rapport de Recherche RR-2772, INRIA, 1996.

[83] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[84] B. Sterzbach. GPS-based clock synchronization in a mobile, distributed real-time system. *Real-Time Syst.*, 12(1):63–75, 1997.

[85] J. Sun. *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems*. PhD thesis, UIUC, 1997.

[86] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[87] Y. Tamir and G. Janakiraman. Hierarchical coherency management for shared virtual memory multicomputers. *Journal of Parallel and Distributed Computing*, 15(4):408–419, 1992.

[88] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3), 1994.

[89] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. 50:117–134, 1994.

[90] Wikipedia. Software transactional memory — wikipedia, the free encyclopedia, 2008. `http://en.wikipedia.org/w/index.php?title=Software_transactional_memory&oldid=213906392`, [Online; accessed 24-May-2008].

[91] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05*, pages 240–248, New York, NY, USA, 2005. ACM.

[92] Y. Zhao, E. A. Lee, and J. Liu. Programming temporally integrated distributed embedded systems. Technical Report UCB/EECS-2006-82, EECS Department, University of California, Berkeley, May 2006.

[93] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.