On collaborative scheduling of distributable real-time threads in dynamic, networked embedded systems

Sherif Fahmy*, Binoy Ravindran*, and E. D. Jensen‡

*ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{fahmy,binoy}@vt.edu

‡The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

Some emerging networked embedded real-time applications have relatively long reaction time magnitudes—e.g., milliseconds to minutes. These longer execution time magnitudes allow opportunities for more computationally expensive scheduling algorithms than what is traditionally considered for device-level real-time control sub-systems. In this paper, we review recent research conducted on collaborative scheduling algorithms in such systems that are subject to dynamic behavior such as transient and sustained resource overloads, arbitrary activity arrivals, and arbitrary node failures and message loss. We show that collaborative scheduling algorithms have an advantage over non-collaborative scheduling algorithms.

1 Introduction

In distributed systems, action and information timeliness is often end-to-end—e.g., a causally dependent, multi-node, sensor to actuator sequential flow of execution in networked embedded systems that control physical processes. Such a causal flow of execution can be caused by a series of nested, remote method invocations. It can also be caused by a series of chained, publication and subscription events, caused due to topical data dependencies—e.g., publication of topic $\mathcal{A}$ depends on subscription of topic $\mathcal{B}$; publication of $\mathcal{B}$, in turn, depends on subscription of topic $\mathcal{C}$, and so on. Designers and users of distributed systems, networked embedded real-time systems in particular, often need to dependably reason about — i.e., specify, manage, and predict — end-to-end timeliness.

Some emerging networked embedded systems are dynamic in the sense that they operate in environments with dynamically uncertain properties (e.g., [1]). These uncertainties include transient and sustained resource overloads (due to context-dependent activity execution times), arbitrary activity arrivals, arbitrary node failures and message loss. Reasoning about end-to-end timeliness is a very difficult and unsolved problem in such dynamic uncertain systems. Another distinguishing feature of motivating applications for this model (e.g., [1]) is their relatively long system reaction time magnitudes—e.g., milliseconds to minutes. Despite the uncertainties, such applications desire the strongest possible assurances on end-to-end activity timeliness behavior.

Maintaining end-to-end properties (e.g., timeliness, connectivity) of a control or information flow requires a model of the flow's locus in space and time that can be reasoned about. Such a model facilitates reasoning about the contention for resources that occur along the flow's locus and resolving those contentions to optimize system-wide end-to-end timeliness. The *distributable thread* programming abstraction which first appeared in the Alpha OS [3] and subsequently in Mach 3.0 [10], and Real-Time CORBA 1.2 [18] provides such a model as their first-class programming and scheduling abstraction. A distributable thread is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote objects.

Deadlines cannot express both urgency and importance. Thus, we consider the *time/utility function* (or TUF) timeliness model [15] that specifies the utility of completing a thread as a function of that thread's completion time. We specify a deadline as a binary-valued, downward "step" shaped TUF. When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the total thread 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 (e.g., [4, 16]).

UA algorithms that maximize total utility under downward step TUFs (e.g., [4, 16]) 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 threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [16] in the sense that the algorithms strive their best to feasibly complete as many high importance threads — as specified by the application through TUFs — as possible.[1] Consequently, high importance threads 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.

In this paper, we consider the problem of scheduling threads in the presence of the aforementioned uncertainties. Thread scheduling approaches can be broadly classified into two major categories, *collaborative* and *independent* scheduling. In the independent scheduling approach, (e.g., [3,5,20]), threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes (thereby not considering node failures during scheduling).

Fault-management is separately addressed by *thread integrity protocols* [11] that run concurrent to thread execution. Thread integrity protocols employ failure detectors (abbreviated here as FDs), and use them to detect failures of the thread abstraction, and to deliver failure-exception notifications [3,5]. In the collaborative scheduling approach (e.g., [19]), nodes explicitly cooperate to construct system-wide thread schedules, anticipating and detecting node failures using FDs.

FDs that are employed in both paradigms in some 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, as pointed out in [13], this results in systems with low coverage. However, 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 [14], Hermant and Widder describe the *Theta-model*, where only the ratio, Θ, between the fastest and slowest message in transit is known. This increases the coverage of algorithms as less assumptions are made about the underlying system. While Θ is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties.

In this paper, we consider a partially synchronous computational model. In particular, we consider the partially synchronous model in [2], where message delay and message loss are probabilistically described. We compare the behavior of three collaborative scheduling algorithms that do not take thread dependencies (that arise due to synchronization, for example) into account (QBUA [7], ACUA [9] and CUA [19]) against an independent scheduling algorithm that also does not consider dependencies (HUA [20]). We also compare the performance of a collaborative scheduling algorithm that considers dependencies (DQBUA [8]) to an independent approach that also does so (RTG-DS [12]).

[1] Note that the term "best effort" as used in the context of networks actually is intended to mean "least effort."

2. Models

Distributable Thread Abstraction. 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*. A thread can be viewed as being composed of a concatenation of thread segments. It 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 sections of a thread are known when it arrives into the system. 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 $\mathbf{T} = \{T_1, T_2, \ldots\}$ and the set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$.

Timeliness Model. A thread's time constraint is expressed using a Time/Utility Function (TUF) [15]. A TUF decouples the urgency of a thread from its importance. This is useful since the urgency of a thread may be orthogonal to its importance. A thread T_i's TUF is denoted as $U_i\left(t\right)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{0, \{m\}\}$. We focus on downward step TUFs, and denote the maximum, constant utility of a TUF $U_i\left(t\right)$, simply as U_i. Each TUF has an initial time I_i, which is the earliest time for which the TUF is defined, and a termination time X_i, which, for a downward step TUF, is its discontinuity point. $U_i\left(t\right) > 0, \forall t \in \left[I_i, X_i\right]$ and $U_i\left(t\right) = 0, \forall t \notin \left[I_i, X_i\right], \forall i$.

System Model. There are actually two system models we consider in this paper. The first makes a distinction between client and server nodes, while the second makes no such distinction. Here we state the properties of the first model, taking into account the fact that the second model is exactly the same, but has no server nodes.

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 [3]), have access to GPS clocks that provides each node with a UTC time-source with nanosecond accuracy (e.g., [21]) and are equipped with appropriately tuned QoS failure detectors [2]

Exceptions and Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for thread failures including time-constraint violations and node failures. If either of these events occur, exception handlers are triggered to restore the system to a safe state. The exception handlers considered have time constraints expressed as relative deadlines.

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, [6] — 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 (see [7]). 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. Note that for the second system model we consider (the one with no server nodes) the discussion above still holds with the exception of the material on servers.

Resource Model. As mentioned before, we overview algorithms that do not consider resource dependencies as well as ones that do. This section only pertains to the latter. Threads can access serially reusable non-CPU resources located at their nodes during their execution. We consider the single resource model — i.e., a thread cannot have more than one outstanding request at any given instance of time. Resources are shared under mutual exclusion constrains and a thread explicitly releases all granted resources before termination. Threads are assumed to access their resources in arbitrary order — i.e., which resources are needed by which threads is not known a priori. Thus we employ deadlock detection and resolution methods instead of prevention and avoidance techniques.

Resource request/release pairs are assumed to be confined within one node, 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. Such a lock is released when the thread's head returns back to the node on which the resource was acquired.

3. Rationale for Collaboration

The main feature of independent scheduling is that each node constructs its local schedule using only local information. This lack of global information makes it impossible for a node to make a globally optimal decision. Thus it is possible for a node to make a scheduling decision that is locally optimal in terms of the utility that can be accrued to the node, but compromises global optimality.

Thus we believe that collaborative scheduling is a better paradigm for systems that can withstand its larger overhead. It should be noted that this is only an issue when the system is overloaded. As explained above, UA algorithms usually default to EDF during underloads. Therefore, both collaborative and independent scheduling algorithms should schedule all threads to completion in this case. During overlaods, however, decisions need to be made about which threads to exclude from the schedule (since, by definition, not all threads can execute to completion). These decisions need to be made consistently on all nodes in a way that minimizes the loss of utility to the system. In addition, if we consider thread dependencies then some sort of collaboration is also needed to resolve the remote dependencies that may occur.

4. Algorithms Overview

We now overview the algorithms we will be comparing in the paper. 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 [4], which uses the heuristic of favoring tasks with a high utility to execution time ratio, i.e., high PUD, when constructing the schedule. The modifications made to the node scheduler ensure that the exception handler of any section included in the schedule is feasible. This modification is made to ensure that when a thread fails the system can be brought back to a safe state. Basically, this is done by inserting both a section and its exception handler into the schedule and then testing it for feasibility. If the schedule is feasible, the section is added to the waiting queue, otherwise both the section and its handler are discarded. This is a low overhead algorithm that guarantees all threads meet their time constraints during underloads. However, we prove in [9] that HUA not have the best effort semantics of DASA in terms of system accrued utility since it only considers local information while constructing the schedule.

CUA is a collaborative scheduling algorithm. It is designed to overcome the shortcomings of independent scheduling algorithms by taking into account global information while constructing a schedule. In CUA, 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 (similar to the one used in HUA). After each node constructs its local schedule, it broadcasts it to all other nodes. The nodes then select a set of threads they consider eligible for execution by removing any thread that has a missing section. Then an instance of distributed consensus is started to reach global agreement on the set of threads to consider for scheduling in the system. CUA is designed for totally synchronous systems and uses local information to construct a local schedule which is then used as an input to an instance of distributed consensus. We prove in [9] that CUA also does not have the same semantics as single node DASA.

In [9], ACUA is developed. ACUA is an improvement on CUA since it is designed for the partially synchronous system model of [2] and thus has greater coverage. In addition, ACUA uses global potential utility density (PUD) of a section (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) to make scheduling decisions on each node. Global PUD allows us to have a total order on the threads in the system in terms of their PUD and thus prevents us from making local decisions that can compromise global optimality. In [9] we present the full algorithm and provide proofs for the properties of ACUA that show that it attempts to mimic the semantics of single node DASA on a distributed system as much as possible. Both CUA and ACUA are based on distributed consensus and thus have high overhead. In order to reduce their overhead, we considered quorum based algorithms.

In [7], we develop QBUA, a quorum based scheduling algorithm for distributable real-time threads in partially synchronous systems. In QBUA, when a node detects a distributed scheduling event (the failure of a node or the arrival of a new thread) it contacts a quorum system requesting permission to run an instance of QBUA (in order to construct a global schedule). All other scheduling events, such as section completion, are dealt with locally.

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 <u>S</u>ystem <u>W</u>ide <u>E</u>xecutable <u>T</u>hread <u>S</u>et (or SWETS), and multicasts any updates to nodes whose schedule has been affected.

The node that computes SWETS does so by running a modified DASA algorithm (similar to the one used in HUA) that orders threads according to their global PUD and then tentatively inserts their sections into the schedule of all the nodes that will be hosting them. After the sections have been inserted, the schedules are tested for feasibility. If all schedules are feasible, the changes are accepted, otherwise all the sections belonging to that thread are removed from the system. We prove in [7] that QBUA has less overhead than ACUA and has the same best effort properties.

In [8],we develop DQBUA, a version of QBUA that handles dependencies. The algorithm is similar to QBUA, but the node that computes SWETS does so while taking into account resource dependencies. The scheduling algorithm in DQBUA first builds the dependency graph for each section by following the set of resource requests and ownership. Then the potential utility of completing both the current thread and its dependents is computed. DQBUA then tests for deadlock by attempting to detect cycles in the dependency graph. Threads are ordered by their potential utility (which includes the utility of the threads they depend on if such threads exist), and DQBUA then tentatively inserts the thread's sections and their dependencies into the schedule in an order that respects dependency constraints. While doing so, a least effort heuristic similar to the one used in DASA is used to reduce the time that a section blocks waiting for a particular resource. We show in [8] that DQBUA has best effort semantics similar to DASA.

In [12], an independent scheduling algorithm, RTG-DS, is presented that schedules dependent threads. RTG-DS is an independent scheduling algorithm in the sense that the scheduling parameters of each section is propagated to each node and the node then constructs a local schedule accordingly. The propagation of section scheduling parameters is accomplished using a gossip based information dissemination algorithm that increases the reliability of the underlying communication layer. The gossip protocol is also used as a method of service discovery to identify the node that will be hosting the next head of the thread. RTG-DS utilizes the slack of each section when constructing a schedule in order to ensure that all other sections have sufficient slack time to gossip in order to find the next head node. Deadlocks are detected by finding cycles in the dependency graph and remote dependencies are resolved by propagating the PUD of local sections to their remote dependents. See [12] for the full algorithm.

5. Experimental Results

We performed a series of simulation experiments on ns-2 [17] to compare the performance of the algorithms summarized in Section 4 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.

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 (and only fifty clients for the algorithms that do not need 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, 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 QBUA. The strength of QBUA 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 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. In [7], we consider other thread sets which produce similar results.

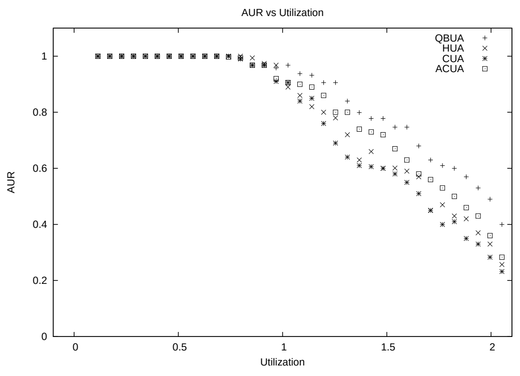

Figure 1. AUR vs. Utilization (no failures)

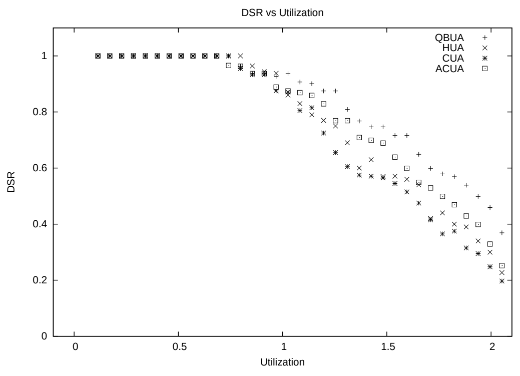

Figure 2. DSR vs. Utilization (no failures)

Figures 1 and 2 show the result of our AUR and DSR experiments in the absence of node failure for the algorithms that do not consider dependencies (HUA, CUA, ACUA and QBUA). As Figures 1 and 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 [7]). 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 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.

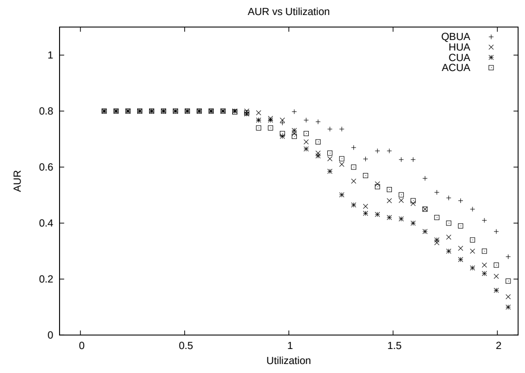

Figure 3. AUR vs. Utilization (failures)

Figure 3 shows the effect of failures on QBUA. In this experiment we programmatically fail $f_{max} = 0.2N$ nodes — i.e., we fail 20% of the client nodes. From Figure 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, now QBUA accrues, on average, 18.5% more utility than CUA, 13.6% more utility than HUA and 9.9% more utility than ACUA. Thus both ACUA and CUA suffer a further loss in performance relative to QBUA in the presence of failures. This occurs because both these algorithms' time complexity is a function of the number of node failures, therefore they have higher overheads in the presence of failures.

We also performed experiments to compare the performance of the algorithms that consider dependencies (RTG-DS and DQBUA). As can be seen in Figures 4 and 5, 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.

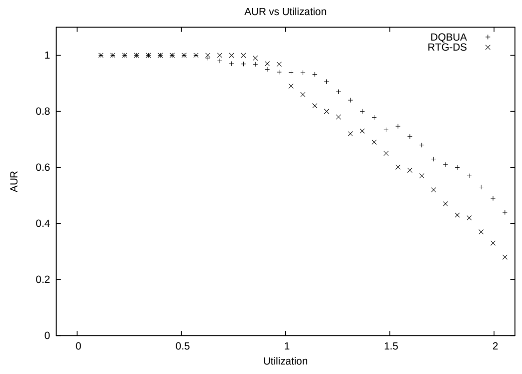

Figure 4. AUR vs. Utilization (considering dependencies)

6. Conclusions and Ongoing Research

In this paper, we presented an overview of recent research on collaborative scheduling in dynamic, networked embedded systems. The result of our study indicates that collaborative scheduling is more beneficial than independent scheduling in systems with relatively large response time magnitudes. Such systems can tolerate the larger overhead involved with collaborative scheduling and thus are in a better position to benefit from their better best effort properties. In addition, since remote dependencies among threads can occur when there are dependency relations among distributable threads, some sort of collaboration is also necessary. This collaboration can be part of the scheduling process (as in [8]) or can occur when resolving resource dependencies (as in [12]).

For systems without remote dependencies, both collaborative and independent scheduling algorithms provide optimal system utility during underloads. Thus, it is better to consider independent scheduling algorithms for such systems since the higher overhead of collaborative scheduling may reduce the amount of user code that can be executed before the system becomes overloaded. During overloads, however, collaborative scheduling provides a considerable advantage over in-

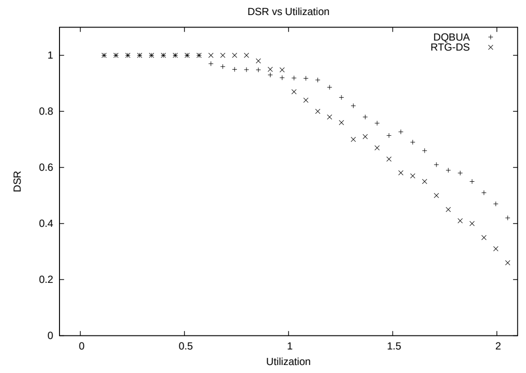

Figure 5. DSR vs. Utilization (considering dependencies)

dependent scheduling for systems with large response times as the experiments above indicate. Theoretical properties of the algorithms compare above can be found in the appropriate references.

Ongoing research involves studying more complex programming abstractions. In particular, enriching the distributable thread abstraction to include information about which threads should never be aborted, which threads should have their orphaned sections continue execution despite node failures etc, is a direction of ongoing research. We also envision developing more sophisticated methods of event notification and studying the effect of different thread call semantics.

Other directions for research include designing algorithms for the fail-recover model, where failed nodes can recover and continue execution. Such failure models pose interesting questions in terms of designing, among other things, real-time persistent storage mechanisms and determining the semantics of orphaned sections when it is possible for failed nodes to recover and hence reconnect the thread. In addition, developing collaborative scheduling algorithms for more dynamic infrastructures such as those that use ad hoc network architectures is a challenging problem.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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`.

[8] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling dependent distributable real-time threads in dynamic networked embedded systems, December 2007. Available at: `http://filebox.vt.edu/users/fahmy/TR-DIPES.pdf`.

[9] 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, Track on Real-Time Systems*, March 2008. To appear, available at: `http://www.real-time.ece.vt.edu/sac-rts2008(%20RTS-115).pdf`.

[10] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *USENIX Technical Conference*, pages 97–114, 1994.

[11] J. Goldberg, I. Greenberg, et al. Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International, January 1995.

[12] 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.

[13] J.-F. Hermant and G. L. Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931 – 944, August 2002.

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

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

[16] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, CMU, 1986. CMU-CS-86-134.

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

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

[19] 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.

[20] 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.

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