FIT: A Flexible, Lightweight, and Real-Time Scheduling System for Wireless Sensor Platforms

Wei Dong, *Student Member, IEEE*, Chun Chen, Xue Liu, *Member, IEEE*,
Kougen Zheng, Rui Chu, and Jiajun Bu, *Member, IEEE*

Abstract—We propose FIT, a flexible, lightweight, and real-time scheduling system for wireless sensor platforms. There are three salient features of FIT. First, its two-tier hierarchical framework supports customizable application-specific scheduling policies, hence, FIT is very *flexible*. Second, FIT is *lightweight* in terms of minimizing the thread number to reduce preemptions and memory consumption while at the same time ensuring system schedulability. We propose a novel Minimum Thread Scheduling Policy (MTSP) exploration algorithm within FIT to achieve this goal. Finally, FIT provides a detailed *real-time schedulability* analysis method to help check if application's temporal requirements can be met. We implemented FIT on MicaZ motes and carried out extensive evaluations. Results demonstrate that FIT is indeed flexible and lightweight for implementing real-time applications, at the same time, the schedulability analysis provided can predict the real-time behavior. FIT is a promising scheduling system for implementing complex real-time applications in sensor networks.

Index Terms—Real-time and embedded systems, hierarchical design, scheduling, wireless sensor networks.

1 Introduction

Recently, Wireless Sensor Networks (WSNs) have seen an explosive growth in both academia and industry [1], [2], [3], [4], [5]. They have received significant attention and are envisioned to support a variety of applications including military surveillance, habitat monitoring, and infrastructure protection, etc.

A WSN typically consists of a large number of microsensor nodes that self-organize into a multihop wireless network. As the complexities for real-world applications (e.g., Redwoods [1], VigilNet [2], etc.) continue to grow, infrastructural support for sensor network applications in the form of system software is becoming increasingly important. The limitation exhibited in sensor hardware and the need to support increasingly complicated and diverse applications have resulted in the need for sophisticated system software. As a large portion of WSN applications is real time in nature [6], a good real-time scheduling system plays a central role in task processing on sensor nodes.

The first emerged sensornet OS, TinyOS [7], is especially designed for resource-constrained sensor nodes. Because of its simplicity in the event-based single-threaded scheduling policy, time-sensitive tasks cannot be handled gracefully in conjunction with other complicated tasks (e.g., compression, aggregation, and signal processing, etc.), as task preemption is not natively supported. Thus, as a programming hack, a long task usually has to be manually split into smaller subtasks to ensure the temporal correctness of the whole system to be met. Otherwise, a critical task could be blocked for too long, hence miss its deadline. Two other notable similar OSs, Contiki [8] and SOS [9], both fall into the same category.

As an alternative solution, Mantis OS [10] borrows time-sliced multithreaded scheduling mechanism from traditional general-purpose OSs. It supports task preemption and blocking I/Os, enabling microsensor nodes to natively interleave complex tasks with time-sensitive tasks. However, Mantis OS is not very flexible and has a relative higher overhead: making changes to its scheduling policy is not an easy task, as the scheduling subsystem is tightly coupled with other subsystems; time-sliced multithreaded scheduling incurs a higher scheduling overhead because it requires more memory to be reserved for thread management.

In this paper, we present a novel scheduling system, FIT, for microsensor platforms. It is flexible through the careful design of its two-tier hierarchical scheduling framework. Under this framework, application programmers can easily implement the most appropriate application-specific scheduling policy by customizing the second-tier schedulers. It is lightweight compared with Mantis OS, as it minimizes the thread number to reduce memory consumption while ensuring schedulability by exploiting the Minimum Thread Scheduling Policy (MTSP) exploration algorithm. In addition, FIT provides detailed real-time schedulability analysis to help the designers to check if application's real-time temporal requirements can be met under FIT's system model.

- *W. Dong, C. Chen, K. Zheng, and J. Bu are with the Zhejiang Key Laboratory of Service Robot, College of Computer Science, Zhejiang University, Zheda Road 38, Hangzhou 310027, China.*
  E-mail: {dongw, chenc, zkg, bjj}@zju.edu.cn
- *X. Liu is with the School of Computer Science, McGill University, Montreal, Quebec H3A 2A7, Canada. E-mail: xueliu@cs.mcgill.ca.*
- *R. Chu is with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Deya Road, Changsha 410073, China. E-mail: rchu@nudt.edu.cn.*

Manuscript received 23 Apr. 2008; revised 25 Nov. 2008; accepted 24 Feb. 2009; published online 3 Mar. 2009.
Recommended for acceptance by C. Qiao.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-04-0148.
Digital Object Identifier no. 10.1109/TPDS.2009.42.

1045-9219/10/$26.00 © 2010 IEEE Published by the IEEE Computer Society

To validate FIT's efficacy and efficiency, we implemented FIT on MicaZ motes and carried out extensive evaluations. Our results show that FIT meets its design objectives. It is flexible as a series of scheduling policies in existing sensornet OSs can easily be incorporated into FIT's two-tier scheduling hierarchy (Section 9.1). It is lightweight as it effectively reduces the running thread number by using the MTSP exploration algorithm (Section 9.2). It is real time as the schedulability analysis is conducted in the MTSP exploration algorithm, thus, real-time guarantees can be achieved by employing the *explored* MTSP (Section 9.3). FIT is a promising scheduling system for developing real-time sensor network applications (e.g., real-time packet transmission, real-time sensing, and processing, etc.).

The main contributions of this paper are summarized as follows:

- We incorporate two-tier hierarchical scheduling into resource-constrained sensor nodes to improve system flexibility and provide good customization of different scheduling policies.
- We perform schedulability analysis of a task that consists multiple run-to-completion jobs. This task definition is more natural to a logical task that contains not only CPU computing time but also resource (i.e., I/O) access time. For this reason, our schedulability analysis is better suited to sensor nodes compared to traditional analysis.
- We employ MTSP to reduce the runtime overheads of threading. To the best of our knowledge, no existing sensornet OSs [11], [8], [9], [10], [12], [13], [14] perform such detailed real-time schedulability analysis and optimizations.

Compared with our preliminary version of this work appeared in [15], this paper includes a number of important extensions as follows:

- We describe FIT's implementation details.
- We prove MTSP's optimality under direct mapping.
- We discuss the FIT's extensibility to distributed systems.
- We describe how we estimate task execution time and I/O service time.
- We evaluate FIT in much more details with TinyOS and Mantis OS.

The rest of this paper is organized as follows: Section 2 describes related work most pertinent to this paper. Section 3 gives an overview of the FIT scheduling system. Section 4 presents our flexible two-tier hierarchical scheduling framework within FIT. Section 5 details the lightweight MTSP exploration algorithm, which relies on the real-time schedulability analysis presented in Section 6. Section 7 discusses FIT's extensibility to distributed systems. Section 8 introduces FIT's implementation, followed by Section 9 that shows the evaluation results. Finally, we conclude the paper and give future directions of work in Section 10.

2 Related Work

FIT borrows heavily from three large areas of prior work: hierarchical scheduling, task scheduling on sensor nodes, and real-time scheduling.

2.1 Hierarchical Scheduling

Hierarchical scheduling techniques have been used in a number of research projects to create flexible real-time systems, such as PShED [16] and HLS [17]. These projects proposed new hybrid algorithms that provide reservation-like guarantees to real-time applications and collections of applications in open environment, where task characteristics are not always known in advance. While their work focuses on general-purpose PCs in open environment, our current work focuses on closed, static, deeply embedded sensor nodes. Regehr et al. [18] describe and analyze the hierarchical priority schedulers already present in essentially all real-time and embedded systems while our current work uses the design philosophy of hierarchical scheduling to implement a two-tier flexible scheduling architecture on resource-constrained sensor nodes.

2.2 Task Scheduling on Sensor Nodes

The event-driven scheme is commonly used in sensornet OSs. TinyOS [7], SOS [9] both fall into this category. In TinyOS, tasks are scheduled in an FIFO manner with a run-to-completion semantic. Like TinyOS, the SOS [9] scheduling policy is also nonpreemptive, and hence, cannot provide good real-time guarantees. Mantis OS [10] uses a different scheme. It supports time-sliced preemptive multithreading. Similar to Mantis OS, Nano-RK [12] provides multithreading support and uses priority-based preemptive scheduling. It performs static schedulability analysis to provide real-time guarantees. However, as mentioned in Section 1, they are not very flexible and still have a relative higher overhead. A large body of work was also devoted to improving the capability of task scheduling based on TinyOS [18], [19], [20], [21].

FIT differs from most existing work in sensornet OSs in many important ways. Its two-tier hierarchical framework supports customizable application-specific scheduling policies, hence, FIT is more flexible. FIT is also lightweight in terms of effectively reducing the running thread number while at the same time ensuring schedulability. Compared with [12], we adopt an event-based programming style, hence it has the opportunity to effectively reduce the running thread number while ensuring schedulability. Compared with [21], FIT automatically assigns tasks to appropriate scheduling queues according to their temporal requirements and provides detailed schedulability analysis.

2.3 Real-Time Scheduling

There is a large body of work devoted to real-time scheduling [22], [23]. Priority mapping and Preemption Threshold Scheduling (PTS) are the most pertinent ones to our work.

Katcher et al. [24] proposed a method to find a best mapping (i.e., with the least schedulability loss) given a limited number of priority levels. Cayssials et al. [25] proposed a method to minimize the priority levels. But they did not present a formal priority assignment algorithm to achieve this end. PTS was first proposed in [26] in order to improve the system schedulability by exploiting nonpreemptiveness as well as reduce the runtime overhead.

Our current work is different from traditional priority mapping in that each task has a two-dimensional priority

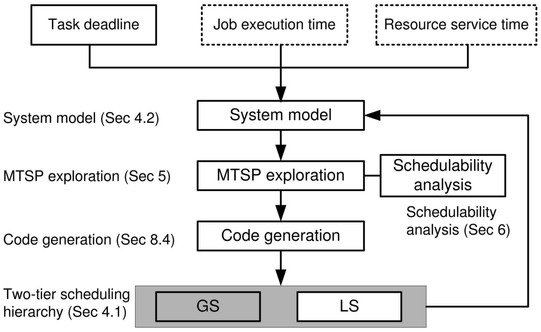

Fig. 1. Design architecture of FIT.

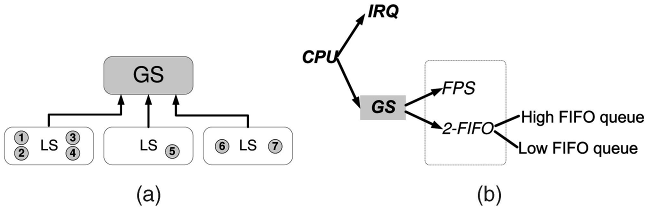

Fig. 2. Two-tier scheduling hierarchy. (a) GS and LSs. (b) An example customization.

(i.e., a global priority and a local priority). Within the same global priority, we differentiate tasks by their local priorities. Hence, with the same number of global priorities (priority levels), there are possibilities that our MTSP exploration algorithm generates feasible assignment that those approaches cannot. We not only try to overlap the global priorities (priority levels) but also try to overlap the local priorities. Our work is also different from PTS as it has no notion of preemption threshold, thus, priorities do not change at runtime. Finally, our work is different from all the above-mentioned ones since a task in FIT consists of multiple jobs with run-to-completion semantics. We analyze schedulability for tasks instead of individual jobs and take into account resource access time, hence, the analysis is better suited to sensornet applications.

3 Overview of FIT

In this section, we give a brief overview of the FIT scheduling system. Details of FIT will be discussed in the following sections.

A block diagram of our FIT scheduling system is shown in Fig. 1. For applications developed under FIT, programmers write applications conforming to event-driven programming style, i.e., run-to-completion semantics and asynchronous I/Os are assumed. By employing Worst Case Execution Time (WCET) estimation techniques [27], [28], we can extract the (worst-case) job execution time and the (worst-case) resource service time from the application (see Section 8.2). These serve as FIT's inputs, which are depicted in the dashed boxes in Fig. 1. In order to attain real-time guarantees, we should carry out schedulability analysis under FIT, hence, task deadlines are also depicted as input parameters. These three inputs provide application-specific data to our system model (see Section 4.2), which is built upon the two-tier scheduling hierarchy (see Section 4.1) of FIT. Then our MTSP exploration algorithm (see Section 5) analyzes the system model and generates the MTSP for the application. The MTSP exploration algorithm relies on real-time schedulability analysis (see Section 6). The resulting MTSP is represented as a priority assignment to each task. It will be further fed into our code generator (see Section 8.4) which will finally generate the corresponding source code to complete our scheduling system.

The two-tier scheduling hierarchy consists of a Global Scheduler (GS) and Local Schedulers (LSs). The first-tier scheduler GS is used to schedule the second-tier schedulers LSs. The GS schedules LSs in a preemptive fashion according to the *global priority* assigned to each LS. The second-tier scheduler LS is used to schedule tasks assigned to it. An LS schedules tasks according to the *local priority* assigned to each task. In FIT, the scheduling policy of an LS must be in a nonpreemptive manner. Thus, each task can be viewed as having two-dimensional priorities, i.e., a global priority and a local priority. Each global priority corresponds to an LS the task is assigned to, and, a task with a higher global priority can always preempt a task with a lower global priority. Within the same global priority, tasks are differentiated by their local priorities. The corresponding LS schedules them in a nonpreemptive manner. The reason we select nonpreemptive scheduling policies for LSs in the design is that they can save context switching time, hence incur less overhead in the second-tier schedulers. While the scheduling policy of the GS is *fixed* for all applications, the number of LSs and each LS's scheduling policy will be *customized* according to different applications, and, they are automatically generated by our code generator in our scheduling system.

4 Flexible Two-Tier Hierarchical Framework

Scheduling policies in current sensornet OSs are usually difficult to customize. Taking TinyOS-1.x, for example, the tight coupling of the FIFO scheduling policy and the nesC programming language makes it hard, if not impossible, to modify. A *flexible* scheduling framework is important for easier customization of different scheduling policies. This framework should cleanly separate from other components in the system, and ideally, provide lower level scheduling mechanisms (as libraries) to reduce the customization overhead to application programmers. Another fact in current sensornet OSs is that different scheduling policies must be employed exclusively. However, application programmers sometimes need to extend the scheduling system without affecting the current system behavior, thus, different scheduling policies may need to coexist. We solve this problem through decomposition of schedulers. We propose a *two-tier scheduling hierarchy* to enhance FIT's flexibility.

4.1 Two-Tier Scheduling Hierarchy

Our two-tier scheduling hierarchy as depicted in Fig. 2a consists of two tiers of schedulers. The first-tier scheduler GS is designed to schedule LSs. The global scheduling policy employed by the GS is *preemptive* priority scheduling. The GS schedules LSs according to their *global priorities*. The second-tier schedulers LSs are designed to schedule individual tasks. Each LS is implemented as one thread and has its own thread context. The local scheduling policy employed by each LS is in a nonpreemptive manner. An LS schedules tasks according to their *local priorities*. Inside the LS, multiple tasks share a common thread context. The local scheduling policy depends on the number of different local priorities it needs to handle. If there is only one local priority, then FIFO (as in TinyOS) is employed. If the number of local priorities is a constant c that is lower than a threshold, then c-FIFO (which manages c FIFO queues of different priorities) is employed. In practice, we select this threshold as 3, similar to that used in the implementation of SOS [9]. The reason is that when the number of local priorities is small, using FIFOs will incur less overhead compared with maintaining a dedicated priority queue. When the number of local priorities is even larger, a priority queue is maintained and Fixed Priority Scheduling (FPS) is employed. The number of LSs and each LS's scheduling policy are customized for different applications.

Fig. 2b illustrates an example customization in our two-tier scheduling hierarchy. In this and subsequent figures, preemptive schedulers are in a bold oblique font; nonpreemptive schedulers in an oblique font, and nonscheduler entities in the hierarchy are in the standard font. In this example, the GS schedules two LSs (i.e., the FPS-LS and the 2-FIFO-LS) preemptively. The FPS-LS has a higher global priority than the 2-FIFO-LS (which manages 2 FIFO queues of different priorities). Thus, any task in the FPS-LS can preempt tasks in the 2-FIFO-LS. The FPS-LS schedules tasks assigned to it nonpreemptively with the FPS scheduling policy. The 2-FIFO-LS has a local high-priority FIFO queue as well as a local low-priority FIFO queue. It schedules tasks in two priority levels and uses FIFO scheduling within one priority level.

FIT's two-tier hierarchical scheduling framework allows each LS to adopt a different scheduling policy from other LSs. An LS receives control of the CPU from the GS. Control can be taken away from the currently running task and given to the GS by the arrival of an interrupt. The GS then passes control down to LSs. As the scheduling policy of the GS is preemptive, it also does context switching if necessary. Thus, the selected LS can take the opportunity and dispatch tasks according to its own scheduling policy. Different local scheduling policies can coexist (as illustrated in Fig. 2b). We also implement low-level scheduling mechanisms as libraries to further reduce the customization overhead for application programmers.

4.2 System Model

We formally define the system model and notations in this section, which will be used in the rest of the paper. As discussed in Section 4.1, in FIT system, each task is assigned a global priority and a local priority. A task with a higher global priority can preempt a task with a lower global priority. Tasks with the same global priority are scheduled by the same LS within which they may have different local priorities. An LS schedules tasks in a nonpreemptive manner. Tasks with both the same global priority and local priority are scheduled in an FIFO manner.

Whenever we say task A has a higher priority than task B, we mean that either task A has a higher global priority than task B or task A has a higher local priority than task B when their global priorities are equal. Tasks A and B have the same priority if and only if they have the same global priority and local priority.

A task consists of several jobs written with a run-to-completion semantic, i.e., they cannot suspend themselves. The basic scheduling unit of our system is a job. Because I/Os must be done in split phases, we assume *request* to be done at the end of a job while *signal* invokes the start of the next job. There is only one active job at any instant within a task.

We formally define the system model as follows. The system Γ consists of a set of n tasks $\tau_1, \ldots, \tau_n$. Each task τ_i is activated by a periodic sequence of events with period T_i and specified a deadline D_i. A task τ_i contains $|\tau_i|$ jobs and each job may not be activated (released for execution) until the request of the preceding job is signaled. We use J_{ij} to denote a job. The first subscript denotes which task the job belongs to and the second subscript denotes the index of the job within the task. A job J_{ij} is characterized by a tuple of $\langle C_{ij}, B_{ij}, G_{ij}, L_{ij}\rangle$. C_{ij} is the worst-case execution time and B_{ij} is the maximum blocking time to access the shared resource requested by the preceding job. It is the time interval from the completion of the preceding job and the start of the current job. G_{ij} is the global priority used and L_{ij} is the local priority used. The blocking time to access a shared resource, B_{ij}, consists of resource service time b_{ij} and resource waiting time b'_{ij}. The resource waiting time b'_{ij} is related to the specific resource scheduling scheme employed.

To summarize, the system model is formally expressed as

$$\begin{aligned}\Gamma &:= \{\langle \tau_1, D_1, T_1\rangle, \ldots, \langle \tau_n, D_n, T_n\rangle\},\\ \tau_i &:= \{J_{i1}, \ldots, J_{i|\tau_i|}\},\\ J_{ij} &:= \langle C_{ij}, B_{ij}, G_{ij}, L_{ij}\rangle.\end{aligned}$$

We make the following assumptions:

- The task deadline is specified no longer than the task period, i.e., $D_i \leq T_i$.
- A good estimation of C_{ij} and b_{ij} is available.
- All jobs within a task share the common period (i.e., jobs within a task are periodic albeit written with a run-to-completion semantic) and have the same global priority and local priority, i.e., $G_{ij} = G_i \wedge L_{ij} = L_i, \forall 1 \leq j \leq |\tau_i|$.

For many practical WSN applications, aperiodic tasks exist. Due to space limit, the schedulability analysis in this paper primarily considers periodic tasks and we do not include the discussion of aperiodic tasks in this paper. Considering general aperiodic tasks in the schedulability analysis is a hard problem. There are some recent advances in this direction. Interested readers are referred to [29], [30], [31] for details.

5 Lightweight MTSP Exploration

In the previous section, we discussed the flexible two-tier hierarchical scheduling framework that can facilitate customizing different scheduling policies. In this section, we propose a method to find appropriate scheduling policies for a specific application. Specifically, we present the MTSP exploration algorithm, which can effectively reduce the running thread number while ensuring system

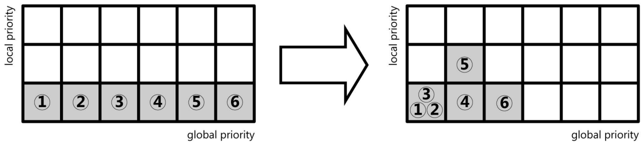

Fig. 3. Global priority and local priority assignment.

schedulability. It is worth noting that we make a distinction between thread and task. Task is from the perspective of functionality while thread is from the perspective of implementation. Thread has implementation and running overhead, e.g., thread context switches, thread control block, thread stack, etc. Traditional general-purpose OSs, including Mantis OS, treat each task as a separate thread. In contrast, our scheduling system tries to overlap multiple tasks so as to reduce the number of threads, and hence the implementation and running overhead, which is important for resource-constrained sensor nodes.

5.1 Problem Formulation

Because the number of global priorities maps directly to the number of LSs (which are implemented as threads) to find the MTSP, we try to minimize the number of global priorities. The constraints are that schedulability of all tasks must be ensured.

We start from a fully preemptive Deadline Monotonic (DM) policy (the left part in Fig. 3). This can be seen as the most capable scheduling policy as DM is optimal in FPS [22] conforming the assumptions under our system model. We call the initially assigned priority as the *natural priority*. Then we try to map the natural priorities to the one with as few global priorities as possible, thus, reducing the running thread number. Hence, preemptions and memory consumption can be reduced, which is important for resource-constrained sensor nodes. Our problem is formally expressed as follows:

Problem. *Given a task set* $\Gamma = \{\tau_1, \ldots, \tau_n\}$ *with increasing natural priority. i.e., natural priority assignment is* $(G_i, L_i) = (i, 0)$. *Find a priority mapping* $(G_i, L_i) : (i, 0) \mapsto (g_i, l_i), 1 \leq g_i, l_i \leq n$, *such that*

$$\begin{aligned} \mathbf{min} \quad & |G|, where\ G = \{g_i\}, \\ \mathbf{s.t.} \quad & all\ tasks\ are\ schedulable. \end{aligned} \tag{1}$$

It should be noted that this problem, similar to the one described in [32], is NP-hard. Hence, it is very important to reduce the time complexity. Otherwise, our MTSP algorithm will have limited value in practice because of scalability issues. To address this issue, we consider finding the optimal solution under the rule of *direct mapping*, which is well studied in the real-time community and effective to reduce the time complexity [32].

Rule (direct mapping). *Assignment ① ↦ Assignment ② is a direct mapping, then, if any task* τ_i *has higher priority than any task* τ_j *in Assignment ①, task* τ_j *cannot have higher priority than that of task* τ_i *in Assignment ②.*

For an illustrative example, let's look at the assignment in Fig. 3. The left part assignment gives a separate global priority to each task. It implies that up to six LSs are needed which correspond to six threads. This is a relatively costly scheme. It may be fine in traditional general-purpose OSs, where CPU speed and memory capacity are abundant. However, for sensor platforms where resources are usually limited, the left assignment is not desirable. Using the MTSP, we can perform static analysis at compile time to effectively reduce the thread number while still ensuring schedulability. The result is shown in the right part. The thread number could be reduced to three.

5.2 MTSP Exploration Algorithm

Our MTSP exploration algorithm is presented in Algorithm 1. After initially assigning a separate global priority to each task, we explore opportunities whether a task could be overlapped with a lower global priority. After iterating for all tasks, we obtain the final result. The time complexity of the MTSP exploration algorithm is hence $O(n)$, i.e., it scales linearly with the number of tasks. From our experience, executing 5-10 tasks (i.e., threads) on current sensor nodes represents quite complex applications. For such task sets, the algorithm runs within a few seconds on a typical PC (e.g., Intel 2.8 GHz CPU, 1 GB DDRII RAM).

Algorithm 1. MTSP exploration algorithm
Input: A task set Γ with decreasing deadlines
Output: Assignment to $(G_i, L_i) = (g_i, l_i)$

```
for $i \leftarrow 1, n$ do
    $(G_i, L_i) \leftarrow (i, 0)$
if test_all_tasks() != TRUE then
    print unschedulable task set
    return
for $i \leftarrow 2, n$ do
    save $G_i, L_i$
    $(G_i, L_i) \leftarrow (G_{i-1}, L_{i-1})$
    if test_task(i) == TRUE then
        continue
    $L_i = L_i + 1$
    if test_task(i) == TRUE then
        continue
    restore $G_i, L_i$
```

Let's revisit Fig. 3 as an example. First, we start with each task assigned with the natural priority. It is worth noting that in this paper, we use a larger value to indicate a higher priority. Then we test if all tasks are schedulable with the most capable scheduling policy, i.e., preemptive DM. If that fails, we report a nonschedulable message and end up with each task assigned to the natural priority. While the task set is schedulable, we examine whether it is still schedulable with a less capable but more lightweight scheduling policy. We start from τ_2. We test whether it is schedulable when assigned with the same global priority and local priority as the previous one (τ_1 in this case). If it is, iterate for the next one (τ_3 in this case). Otherwise, we test whether τ_2 is schedulable when assigned with a higher local priority and with the same global priority as the last one τ_1. If it is, iterate for the next task. If τ_2 is unschedulable in both of the cases, we leave its global priority and local priority unchanged and iterate for the next task. After iterating for all of the tasks, we end up with the resulting global priorities and local priorities while ensuring schedulability.

In Algorithm 1, note that we only test the schedulability of τ_i in the loop. This is ensured by the following theorem.

Theorem 1. *When τ_i moves to a lower priority, the schedulability of all tasks is ensured as long as τ_i is schedulable.*

Proof. We consider the interference caused by τ_i to any other task τ_j. $\forall \tau_j$, the interference caused by τ_i will not increase after the move. Thus, the schedulability of any other task τ_j is ensured and the schedulability of all tasks is ensured as long as τ_i's schedulability is ensured after the move. □

Under the rule of direct mapping, the MTSP exploration algorithm is optimal in the sense that for a given task set and a given number of global priorities, if the task set can feasibly be scheduled using the MTSP exploration algorithm, the number of global priorities required is minimal; if there is an algorithm making the task set schedulable, then the MTSP exploration algorithm can also make it schedulable.

Theorem 2. *Given a schedulable task set, the MTSP exploration algorithm generates a minimum number of global priorities under the rule of direct mapping.*

Proof. Assume that the number of global priorities generated by MTSP is p. If another algorithm ② exists, making Γ schedulable and the number of global priorities is $q < p$, then there is at least one global priority i in which tasks must be repartitioned into other global priorities. If the first task of i, denoted by τ_i, is mapped to $i-1$, the system will be unschedulable. Otherwise, it will be mapped to $i-1$ rather than i using Algorithm 1. If τ_i is mapped to $i+1$, the system will also be unschedulable because of a certain task of $i+1$, denoted by τ_{i+1}. Otherwise, τ_{i+1} will be mapped to i rather than $i+1$ using Algorithm 1. It is in contradiction with the assumption that algorithm ② can make Γ schedulable. □

Theorem 3 (Optimality). *Given a schedulable task set, if there is any direct mapping under which the system is schedulable, then the system is also schedulable under the MTSP exploration algorithm.*

Proof. The approach we take to prove this theorem is to assume that some schedulable direct mapping exists, and to show that we can derive a schedulable MTSP.

Let us assume that some direct mapping exists. Take the lowest natural priority task τ that is assigned to global priority 1 and overlap it into the lower global priority 0. According to Theorem 1, the system will be schedulable iff τ remains schedulable. We repeat this procedure moving the next lowest natural priority task from global priority 1 to global priority 0, as long as the moved task remains schedulable. Clearly, if we continue this procedure for all the tasks, the resulting mapping will be the one that is resulted from using the MTSP exploration algorithm. Hence, the theorem holds. □

6 Real-Time Schedulability Analysis

In this section, we consider the real-time constraints presented in the minimization problem in Section 5.1. It is worth mentioning that our schedulability analysis is conducted on tasks instead of individual jobs. To derive the schedulability test, we analyze the processor demand [22] of each task. In FIT, the processor demand of a task consists of the following:

- C_i: the execution time of τ_i.
- I_{lp}: the interference time from lower priority tasks.
- I_{sp}: the interference time from the same priority tasks.
- I_{hp}: the interference time from higher priority tasks.
- B_i: the blocking time to access shared resources.

A sufficient condition to make τ_i schedulable is [22]:

$$\min_{0<t\le D_i} \frac{W_i(t)}{t} \le 1, \quad \text{where } W_i(t) = C_i + I_{lp} + I_{sp} + I_{hp} + B_i. \tag{2}$$

Note that $C_i, I_{lp}, I_{sp}, I_{hp}$, and B_i may depend on t. When there is no ambiguity occurred, we omit it here and in the following sections for the simplicity of notations.

It is also worth noting that in order to find the minimal of $W_i(t)/t$ over all possible t, we check all discrete t that is a multiple of any task's period, i.e., $t \in \{t = nT_k | 0 < t \le D_i, \tau_k \in \Gamma, n \in N_+\}$. It is because the processor demand of a task $W_i(t)$ is a step function of t and only increases at these discrete points [22].

6.1 Determining C_i

C_i is τ_i's execution time, which equals the sum of the execution time of all jobs within τ_i, i.e.,

$$C_i = \sum_{j=1}^{|\tau_i|} C_{ij}. \tag{3}$$

Note that the job execution time C_{ij} is estimated using existing WCET estimation tools for MicaZ motes (see Section 8.2).

6.2 Determining I_{lp}

I_{lp} is the maximum interference time caused by lower priority tasks. In FIT, as lower global priority tasks can always be preempted by τ_i, they can never block the execution of τ_i. Thus, they introduce no interference. Tasks with the same global priority but with lower local priorities, however, can block the execution of τ_i as they are scheduled nonpreemptively with τ_i. We denote $lp(i)$ as the task set in which tasks have the same global priority but have lower local priority. $lp(i)$ represents the lower priority tasks, which actually cause interferences.

The maximum interference occurs when there is a lower priority task executing each time a job in τ_i releases. τ_i can be at most blocked for $|\tau_i|$ times as there are $|\tau_i|$ run-to-completion jobs. The blocking time each time will not exceed the *maximum* execution time of all the lower priority jobs. Hence,

$$I_{lp} \le \max_{\substack{k \in lp(i) \\ 1 \le j \le |\tau_k|}} \{C_{kj}\} \cdot |\tau_i|.$$

It is worth noting that a tighter bound exists because the time of interference caused by a job J_{kj} is limited to $\lceil t/T_k \rceil$. That means the job with the maximum execution time is not able to block τ_i for $|\tau_i|$ times if $\lceil t/T_k \rceil < |\tau_i|$. We sort $\{C_{kj}\}$, for *all* $k \in lp(i), 1 \le j \le |\tau_i|$, in nonincreasing order, i.e., $C_{k_1j_1} \ge C_{k_2j_2} \ge \cdots \ge C_{k_mj_m} \ge \cdots$, then,

$$I_{lp} \le \underbrace{C_{k_1j_1}\left\lceil \frac{t}{T_{k_1}} \right\rceil + C_{k_2j_2}\left\lceil \frac{t}{T_{k_2}} \right\rceil + \cdots + C_{k_mj_m} + \cdots + C_{k_mj_m}}_{\text{There are total } |\tau_i| \text{ number of } C_{kj}}. \tag{4}$$

We end up with either $|\tau_i|$ number of C_{kj} are added up or we have added all the terms under consideration.

6.3 Determining I_{sp}

I_{sp} is the maximum interference time caused by the same priority tasks. All the same priority tasks have opportunities to interfere the execution of τ_i. We denote $sp(i)$ as the task set in which tasks have the same priority as τ_i.

In FIT, tasks in $sp(i)$ are scheduled in an FIFO manner with τ_i. As with the computation of I_{lp}, τ_i can be blocked at most $|\tau_i|$ times. The blocking time each time will not exceed *all* jobs ahead of τ_i in the same priority FIFO. As there is only one active job among tasks, the blocking time each time is at most $\sum_k \max_j\{C_{kj}\}$, where $k \in sp(i), 1 \le j \le |\tau_k|$. Hence,

$$I_{sp} \le \left(\sum_{k \in sp(i)} \max_{1 \le j \le |\tau_k|} \{C_{kj}\} \right) \cdot |\tau_i|.$$

As with the same reasoning in Section 6.2, a job J_{kj} may not be able to block τ_i for $|\tau_i|$ times because it is further limited by $\lceil t/T_k \rceil$. For *each* $k \in sp(i), 1 \le j \le |\tau_k|$, we sort $\{C_{kj}\}$ in nonincreasing order, i.e., $C_{kj_1} \ge C_{kj_2} \ge \cdots \ge C_{kj_{|\tau_k|}}$. The interference caused by τ_k is

$$I_k \le \underbrace{C_{kj_1}\left\lceil \frac{t}{T_k} \right\rceil + C_{kj_2}\left\lceil \frac{t}{T_k} \right\rceil + \cdots + C_{kj_m} + \cdots + C_{kj_m}}_{\text{There are total } |\tau_i| \text{ number of } C_{kj}}.$$

Also, we end up with either $|\tau_i|$ number of C_{kj} are added up or we have added all the terms under consideration. The overall interference is then

$$I_{sp} \le \sum_{k \in sp(i)} I_k. \tag{5}$$

6.4 Determining I_{hp}

I_{hp} is the maximum interference caused by higher priority tasks. As all higher priority tasks can interfere the execution of τ_i, we denote $hp(i)$ as the task set in which tasks have higher priority than τ_i. The processor demand of higher priority tasks is limited by

$$I_{hp} \le \sum_{k \in hp(i)} C_k \lceil t/T_k \rceil, \tag{6}$$

where C_k is given by (3).

6.5 Determining B_i

τ_i blocks $|\tau_i| - 1$ times (assume that there is no blocking for the first job to execute) to access shared resources. As discussed in Section 4.2, B_{ij} denotes the maximum blocking time to access the shared resource requested by the preceding job $J_{i(j-1)}$. Each blocking time B_{ij} consists of resource service time b_{ij} and resource waiting time b'_{ij}. Hence,

$$B_i = \sum_{j=2}^{|\tau_i|} B_{ij} = \sum_{j=2}^{|\tau_i|} (b_{ij} + b'_{ij}). \tag{7}$$

The resource service time b_{ij} is estimated beforehand (see Section 8.2) while the resource waiting time depends on the resource scheduling scheme. We denote $rc(i,j)$ as the task set in which tasks also use the same resource as that J_{ij} blocks on (before execution). Further, we introduce $rc_m(i,j)$ to represent all the jobs in task m, that $\forall k \in rc_m(i,j), J_{mk}$ blocks on the same resource as that J_{ij} blocks on. If FIFO resource scheduling scheme is employed, then

$$b'_{ij} \le \sum_{m \in rc(i,j)} \max_{k \in rc_m(i,j)} \{b_{mk}\}.$$

7 Extention

It is worth mentioning that this paper primarily considers real-time task scheduling on a single node. However, FIT's task model can be extended to distributed networking systems. It is feasible because we perform an end-to-end schedulability analysis for a multiple of jobs, which could potentially be located at different nodes. According to a similar procedure as described in [32], we demonstrate how FIT's approach can be applied using the concrete steps as listed below.

Initial assignment. The algorithm assigns separate global priorities to all tasks (including global tasks that consist of jobs residing on different nodes) according to DM.

Schedulability analysis. The algorithm next performs schedulability analysis on the tasks. The distributed schedulability analysis procedure has two extensions. First, when considering the inference from other tasks for task τ_i, we separately consider the interference for each job (within task τ_i) at a single node the job resides on. This is because only jobs at the same node contend with each other. Then, we sum up interferences to all the jobs (within task τ_i) to get the interference for task τ_i. Second, the communication link is also modeled as a shared resource and its resource service time is defined to be the maximum MAC delay (we assume a TDMA schedule at the MAC layer to avoid transmission collisions). With this distributed schedulability analysis procedure, we test the schedulability of the whole system. If the system is schedulable, it goes to the next step. If the system is not schedulable, then the algorithm quits since no mapping will ever improve the schedulability of the system.

Scan and overlap. The goal here is to overlap as many as necessary into fewest number of global priorities at each node without making the system unschedulable. The algorithm scans the tasks in increasing global priority order, trying to overlap the currently scanned task into a lower priority. The distributed mapping procedure has one distinction, which is worth noting. That is, within the currently scanned task, we consider moving each job (in their execution order) to a lower priority on the node the job resides on. The reason is that this strategy may reduce the global priority number at certain nodes. After each moving step, the algorithm tests the schedulability of the task. If it is found to be schedulable, the priorities of all jobs within the task are changed accordingly and the scan goes on to the next moving step (i.e., for the next job in execution time or the first job within the next higher global priority task). If the

TABLE 1
APIs for the GS

Prototype	Description
`sched_init()`	Initialize the scheduling system
`sched()`	Schedule jobs whenever necessary
`post(JOB*)`	Post a job to the appropriate scheduling queue
`ls_new(uint8_t gp, uint8_t(*post)(JOB*), void(*start)(void))`	Create an LS whose global priority is `gp`. The prototypes of `post` and `start` are implemented by an LS.
`ls_destroy(uint8_t gp)`	Destroy the LS whose global priority is `gp`

`CONTEXT_INIT()`	Initialize the execution context of an LS
`CONTEXT_SAVE()`	Save the execution context of a LS
`CONTEXT_RESTORE()`	Restore the execution context of an LS

task is found to be nonschedulable, the algorithm quits and we left with the priorities making the system schedulable.

8 IMPLEMENTATION

We have implemented our scheduling system on the MicaZ platform, which has 7.37 MHz Atmega128L microcontroller, with 128 KB program flash, 4 K data RAM, Chipcon CC2420 radio, and MTS310 multisensor boards.

In the following, we select and discuss some important aspects of the implementation.

8.1 Two-Tier Hierarchical Framework

The implementation of FIT's two-tier hierarchical scheduling framework is divided into two parts: the GS implementation and the LS implementation. The GS implementation is fixed for all applications. It schedules LSs preemptively according to their global priorities. We summarize the APIs for the GS in Table 1. `ls_new()` is used to create an LS with the specified global priority. Two function pointers, i.e., `post` and `start`, are passed in as two parameters, where `post` is used to post a job, and `start` is used to dispatch tasks in the LS. The platform-dependent operations are all related to manipulations on the execution context of LSs, which are listed in the bottom-half of Table 1.

The number of LSs and each LS's scheduling policy, however, are customized according to application requirements. Table 2 gives an example LS implementation for the FIFO scheduling policy. We implement prototypes of `post` and `start`, i.e., `fifo_post()` and `fifo_start()` in this case. Once the FIFO-LS gains the CPU control in `fifo_start`, it applies its local scheduling policy. The number of LSs and their corresponding policies are identified by the MTSP exploration algorithm and the source code is generated by our code generator.

TABLE 2
APIs for the FIFO-LS

Prototype	Description
`void fifo_init(void)`	Initialize the FIFO-LS
`uint8_t fifo_post(JOB*)`	Post a job to the specified queue of the FIFO-LS
`void fifo_start(void)`	The FIFO-LS gains the CPU control

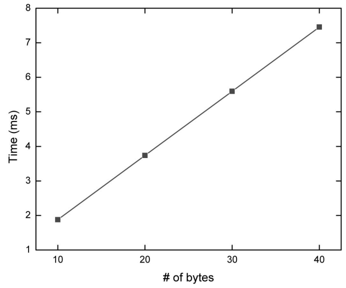

Fig. 4. Resource service time for MicaZ's serial port.

8.2 Timing Analysis

In order to make our MTSP exploration algorithm work, it is important to estimate the job execution time (i.e., C_{ij} as described in Section 4.2) and the resource service time (i.e., b_{ij} as described in Section 4.2) accurately.

To obtain the job execution time, we use existing WCET estimation techniques [27]. For the MicaZ platform with Atmega128L processor, there are existing tools to provide such estimation accurately [28].

It is worth noting that to ensure the safe execution of hard real-time tasks, it is necessary to consider safe (and tight) WCET bounds rather than the average WCET that is unsafe or constrained to probabilistic analysis [28]. Scheduling based on safe WCET may consume more RAM under *certain* conditions, but it is required for meeting the time constraints under *all the possible* conditions. Generally speaking, there is an inherent trade-off between real-time guarantees and resource consumptions. We do allow further reducing resource consumption via programmer assistance. Our algorithm generates scheduler construction code for execution on motes. Based on the assigned priorities, application programmers can further overlap tasks to the same global priority to reduce RAM consumption. In this case, programmers must be aware that the task moved to a lower priority may potentially miss its deadline. It is suitable when the task that is moved to a lower priority could suffer amount of computational delays.

To obtain the resource service time, we quantify the clock cycles to perform common I/O operations on MicaZ under FIT. As Fig. 4 illustrates, the resource service time scales linearly with the number of bytes for I/O and can be well estimated for a given system implementation.

8.3 MTSP Exploration

MTSP exploration is used to determine how many LSs are needed and what are their corresponding local scheduling policies. The number of LSs equals the number of global priorities and the local scheduling policy depends on the number of different local priorities within the global priority. If there is only one local priority, then FIFO (as in TinyOS) is employed. If the number of local priorities is a constant $c \leq 3$, then c-FIFO is employed. When the number of local priorities is even larger, a priority queue is maintained and FPS is employed (discussed in Section 4.1).

We have implemented the MTSP exploration algorithm (together with our code generator that will be described in the following section) in Java. Its implementation is outlined in Algorithm 1. The input of the algorithm is the application task set Γ with specified job execution time, resource service time, and task deadlines.

The output of the algorithm is a priority assignment for each task which is used to carry out customizations for LSs.

8.4 Code Generation

As the final step, the code generator converts the output of the MTSP exploration algorithm, i.e., the priority assignment, into source code which is then incorporated into FIT's scheduling framework. The code generator has two major tasks as follows:

- Constructing LSs using `ls_new();`
- Posting jobs to the correct scheduling queue.

As illustrated in Fig. 3, when the result of the MTSP exploration algorithm is as shown in the right part, the LS construction code using our code generator will be

```
// for task 1, 2, 3;
ls_new(1, ls1_fifo_post, ls1_fifo_start);
// for task 4, 5;
ls_new(2, ls2_2fifo_post, ls2_2fifo_start);
// for task 6;
ls_new(3, ls3_fifo_post, ls3_fifo_start).
```

9 Evaluation

We implemented FIT on MicaZ motes and carried out extensive tests to evaluate the flexibility, lightweightness, and real-time performance of FIT. First, we examine whether FIT is flexible enough to incorporate scheduling policies in existing sensornet OSs. Second, by a case study of typical workloads on sensor nodes, we examine whether FIT can effectively reduce the running thread number to reduce preemptions and memory consumption. Third, we examine the real-time performance of FIT. Finally, we evaluate FIT's implementation overheads compared to TinyOS and Mantis OS.

9.1 Flexibility

Fig. 5 depicts four existing scheduling policies in sensornet OSs under FIT's two-tier scheduling hierarchy. Fig. 5a shows the two-tier scheduling hierarchy for TinyOS [7]. It has only one LS, i.e., the FIFO-LS. Whenever the GS gains the CPU control to dispatch jobs, it always passes the control down to the FIFO-LS, which schedules tasks in a nonpreemptive FIFO manner. Fig. 5b shows the two-tier scheduling hierarchy for SOS. Also, it has one LS, i.e., the 3-FIFO-LS, which manages three FIFO queues of high, system, and low priorities, respectively [9]. The scheduling hierarchy of Mantis OS [10] is shown in Fig. 5c. It has five LSs with different global priorities. Each LS, different from the previous two, is a *preemptive* round-robin scheduler. Although as discussed in Section 4.1, each LS is assumed as a nonpreemptive scheduler, FIT's two-tier scheduling hierarchy is general and does allow each LS to employ arbitrary scheduling policy. The real-time schedulability analysis and the MTSP exploration algorithm, however, need to be revised once this assumption is broken. Fig. 5d shows the scheduling hierarchy of the PL scheduler [21] for TinyOS-2.x. The GS schedules three LSs in a preemptive manner. The 3-FIFO-LS manages three FIFO queues of different local priorities and schedules tasks in a nonpreemptive manner. As we can see in Fig. 5, FIT's two-tier scheduling hierarchy is indeed flexible as a series of existing scheduling policies can easily be implemented under this framework.

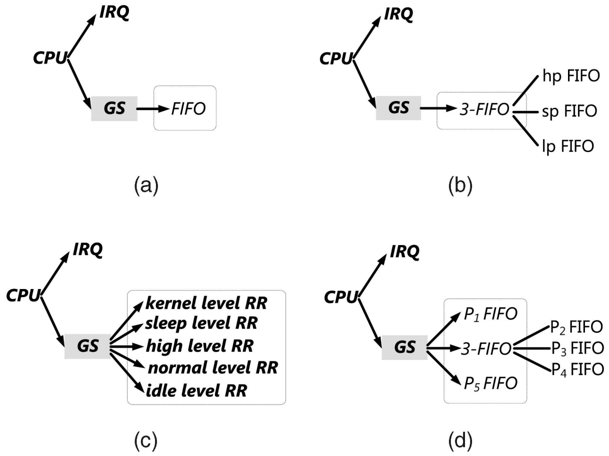

Fig. 5. Existing scheduling policies under FIT's two-tier scheduling hierarchy. (a) Two-tier scheduling hierarchy for TinyOS. (b) Two-tier scheduling hierarchy for SOS. (c) Two-tier scheduling hierarchy for Mantis OS. (d) Two-tier scheduling hierarchy for the PL scheduler.

9.2 Lightweightness

A typical metric to evaluate FIT's lightweightness is its resource usage. Specifically, RAM usage (for data) and program flash (for code) are two important metrics as they are restricted for current mote platforms. For program flash usage, we compare machine code sizes for TinyOS, Mantis OS, and FIT (see Section 9.4). For RAM usage, we have chosen the metric of the number of threads because of two reasons. First, number of threads impacts RAM usage to the largest extent. For example, in Mantis OS, a thread occupies a stack of size 128 bytes. This is relatively large compared to a total of 4 K RAM on MicaZ motes. Second, compared to overall RAM usage, this metric is more suitable because it does not involve system overheads other than the scheduling system. In the following, we examine how FIT reduces the number of threads.

We set up experiments to generate random inputs to our MTSP exploration algorithm to study how many threads are actually needed to ensure the system's schedulability. The parameters we used are as follows. The system consists of $|\Gamma| = 5$ number of tasks where all tasks fall into two categories: long executing tasks with $C_{ij} \sim U(250, 350)$ and real-time tasks with $C_{ij} \sim U(2, 18)$. In addition, real-time tasks have a more urgent deadline than long executing tasks. Each task contains 1-5 number of jobs. The default number of shared resources is $|R| = 4$ and the resource service time varies from 1 to 24 ms. Meanwhile, FIFO resource scheduling scheme is assumed.

We are interested to see how many threads are required as the average percentage of real-time tasks increases. We generate 1,000 cases of task sets for each percentage. Note

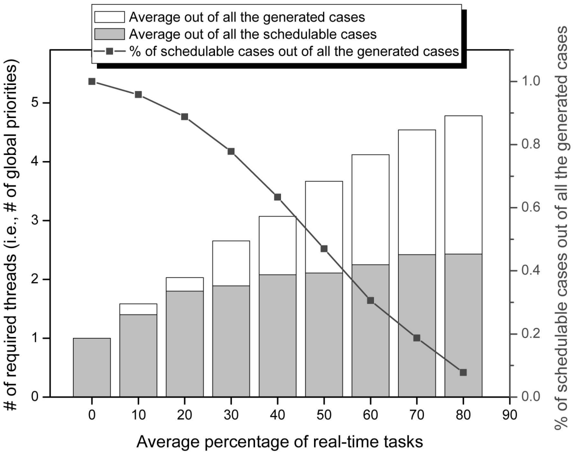

Fig. 6. Number of required threads.

that some of the task sets might be schedulable while others are not. The average results are shown in Fig. 6. The solid line represents the percentage of schedulable cases out of all the generated cases. The white bar represents the average number of threads out of all the generated cases while the gray bar represents the average number of threads out of all the schedulable cases. As we can see from the solid line, as the average percentage of real-time tasks increases, the percentage of schedulable cases out of all the generated cases decreases. Our MTSP exploration algorithm ends up with the maximum number of threads when it encounters an unschedulable case. Thus, the average number of threads out of all the generated cases is approaching $|\Gamma| = 5$ when the average percentage of real-time tasks reaches 80 percent as the majority of the cases are unschedulable. To get a closer look at the cases when the task sets are schedulable, as shown by the gray bar in Fig. 6, we observe that the average number of threads out of all the schedulable cases never exceeds three, reducing at least two threads compared with the worst case.

9.3 Real-Time Guarantee

We show FIT's real-time guarantee via a case study. The system we studied has two shared resources, R_A and R_B, and their individual service time is 1 ms (R_A) and 18 ms (R_B), respectively. The system consists of the following tasks:

- Task τ_1 that consists of four jobs. The individual job execution times are selected as 3, 4, 5, and 3 ms, respectively. The shared resources accessed between consecutive jobs are chosen to be R_A, R_B, and R_A. Task τ_1's period (T_1) and deadline (D_4) are set to 1 s, i.e., $T_1 = D_1 = 1$ s.
- Task τ_2 with the same setting as τ_1.
- Task τ_3 with the same setting as τ_1.
- Task τ_4 that consists of three jobs. The individual job execution times are selected as 3, 4, and 18 ms, respectively. The shared resources accessed between consecutive jobs are chosen to be R_B and R_A. This is the task under consideration and we vary $T_4 = D_4$ from 20 to 160 ms.
- Task τ_5 that consists of three jobs. The individual job execution times are selected as 3, 36, and 524 ms. The shared resources accessed between consecutive jobs are chosen to be R_A and R_B. $T_5 = D_5 = 10$ s.

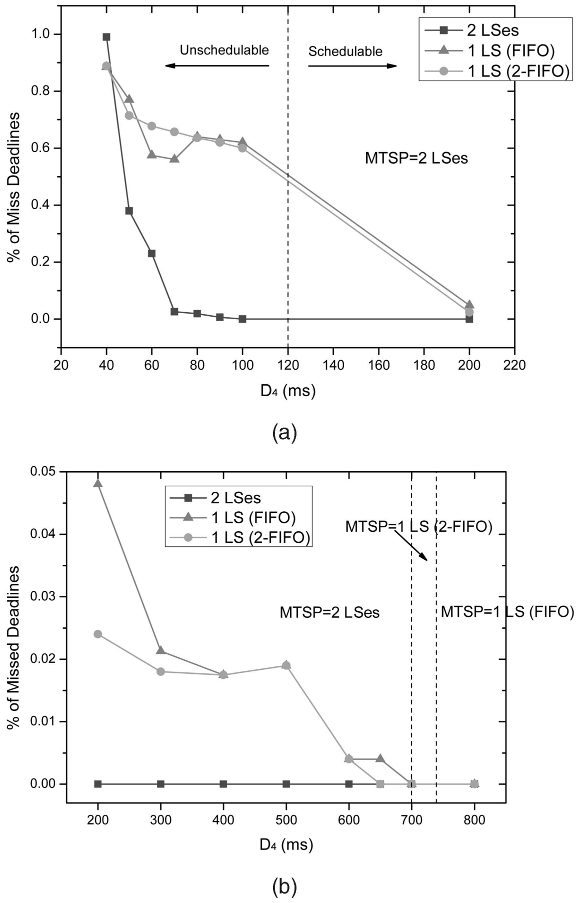

Fig. 7. Varying D_4 from 40 to 800 ms. (a) Varying D_4 from 40 to 200 ms. (b) Varying D_4 from 200 to 800 ms.

Figs. 7a and 7b illustrate the cases when we vary the deadline D_4 within 40-200 ms and 200-800 ms, respectively.

We can see in Fig. 7a that from the theoretical analysis discussed in Section 6, the system will be schedulable when D_4 reaches 120 ms. When $D_4 < 120$ ms, there could be missed deadlines from the theoretical analysis. The results collected from testbed show that when $D_4 \geq 100$ ms, employing the MTSP (with two LSs) will result in no missed deadlines in practice. The gap (between 120 and 100 ms) exists because the theoretical analysis considers the *worst* case, while in practice, the *runtime* interference could be smaller, thus, mitigating the percentage of missed deadlines even in the face of an unschedulable system. When $D_4 < 120$ ms, FIT's real-time schedulability analysis will also report τ_4 as the unschedulable task, thus, designers can take various ways (e.g., relaxing its deadline, reducing concurrency, etc.) to redesign a schedulable system. Fig. 7a also indicates that using the theoretical analysis, when $D_4 \geq$ 120 ms FIT can ensure that there are no missed deadlines by employing the MTSP while the simple FIFO or 2-FIFO scheduling policy may produce missed deadlines in practice. Also note that the percentage of missed deadlines may increase in Fig. 7a. This is because as $D_4 (= T_4)$ increases, the total number of jobs released is reduced.

TABLE 3
Spatial Overheads Comparisons

	TOS-1.x	TOS-2.x	Mantis	FIT-E	FIT-T
Machine code size (bytes)	1530	2206	17224	5180	6192

Fig. 7b shows when $D_4 \geq 700$ ms, from the theoretical analysis, the system will be schedulable with a 2-FIFO scheme, and, when $D_4 \geq 740$ ms, the system will be schedulable with an FIFO scheme. In practice, this transitional region (700-740 ms) is located with a smaller D_4 (650-700 ms) because runtime interference is smaller than worst-case predictions. As Figs. 7a and 7b indicate, FIT always selects a lightweight scheduling policy while at the same time ensuring the schedulability of the system.

9.4 Implementation Overheads

To evaluate FIT's implementation overheads, we first examine the spatial overheads of FIT in terms of machine code size. Next, we examine FIT's temporal overheads. Finally, we conduct a comparative evaluation between TinyOS, Mantis OS, and FIT in terms of CPU utilization using two typical benchmarks.

First, to compare the spatial overhead (i.e., machine code size), we use a GNU tool, avr-size, to count the binary code size of the Blink benchmark under TinyOS, Mantis OS, and FIT. We select this benchmark because of two reasons. First, the Blink benchmark is available for all the three systems (it is named blink_led in Mantis OS). Second, the Blink benchmark is suitable for comparing task scheduling overheads as it does not involve other system services. Under FIT, we implement an event-driven scheduling system as in TinyOS, denoted by FIT-E; and we also implement a multithreaded scheduling system, denoted by FIT-T.

Table 3 reports the results. As we can see that TinyOS (both versions 1 and 2) has the smallest code complexity because the basic scheduling system is relatively simple. Besides, TinyOS and nesC [33] do full program analysis and optimizations, which leads to a very compact single image. Mantis OS borrows code from traditional multithreaded systems and uses dynamic memory allocation. Hence, it is far more complex than TinyOS. FIT's current implementation on MicaZ has larger code size than TinyOS and smaller code size than Mantis OS. One reason that FIT's code complexity is relatively low is that the complex MTSP algorithm is executed at the PC side and the final generated code is optimized for execution on motes. As expected, FIT-T has larger code size than FIT-E as the code for multithreading is more complex. Finally, it is worth mentioning that the machine code sizes for all the systems are acceptable for the MicaZ platform which has 128 K program flash.

Next, to examine the temporal overhead of FIT's two-tier scheduling, we select four metrics as follows:

- overhead of posting a job;
- overhead of scheduling jobs in the same task which is measured as the time interval from the completion of the preceding job to the start of the current job without considering the blocking time to access shared resources for simplicity;
- overhead of LS creation, which corresponds to thread creation in Mantis OS; and
- overhead of scheduling jobs in different LSs, which incurs context switching.

TABLE 4
Temporal Overheads Comparisons (in Clock Cycles)

Operations	TinyOS	Mantis	FIT
Posting a job	42	N/A	90
Scheduling jobs in the same LS	63	N/A	130
LS/thread creation	N/A	2481	366
Scheduling jobs in different LSes	N/A	447	409

We measure the clock cycles of each operation by using Avrora [34]. The results are reported in Table 4. The postoperation of TinyOS is 42 cycles while FIT has 90 cycles. The extra overhead lies in the fact that we make scheduling decisions in our postoperation while TinyOS does not need to as priority preemption is not allowed between TinyOS tasks. When just the postoperation is measured, FIT takes about 44 cycles, which is very close to that of TinyOS. Scheduling jobs in the same LS, however, consumes more time than TinyOS because FIT will pass the CPU control to the GS whenever a job returns. Then the GS passes the CPU control down to the LS, which schedules the next job. In FIT, LS scheduling is about 55 cycles which is close to TinyOS; GS scheduling consumes about 67 cycles, which is an extra overhead introduced by our two-tier hierarchical scheduling. LS/thread creation in FIT consumes 366 cycles, as opposed to 2,481 cycles in Mantis OS. The significant difference stems from the fact that Mantis OS uses dynamic memory allocation, which consumes about 1,655 cycles as well as invoking a thread dispatching routine that consumes about 447 cycles. Without considering these overheads, it leaves 379 cycles, a little larger than that of FIT. In FIT, scheduling jobs in different LSs involves context switching, which consumes 409 cycles, much larger than scheduling jobs in the same LS, but slightly smaller than Mantis OS. As we can see, FIT's flexible two-tier hierarchical scheduling scheme has acceptable temporal overheads.

Finally, we conduct a comparative evaluation among TinyOS, Mantis OS, and FIT in terms of CPU utilization using two typical benchmarks.

Table 5 reports the results for the Blink benchmark. We have compared among TinyOS, Mantis OS, FIT-E (event-driven scheduling under FIT), and FIT-M (the implementation of FIT we port to Mantis OS). For event-driven scheduling, FIT has a 0.0231 percent increase in CPU utilization. For multithreaded scheduling, FIT has a 0.0706 percent increase in utilization. This is because two levels of scheduling are involved in FIT. Considering the fact that CPU utilizations are typically quite low for many benchmarks for WSNs [35], this small difference in CPU utilization will not significantly impact the energy efficiency of WSNs as CPU energy consumption is dominated by energy consumption in the idle mode (which lasts for vast majority of the time).

TABLE 5
CPU Utilization Comparisons for the Blink Benchmark

TinyOS	Mantis	FIT-E	FIT-M
0.0273%	0.5482%	0.0504%	0.6188%

TABLE 6
CPU Utilization Comparisons for the CntToRfm Benchmark

TinyOS-1.x	TinyOS-2.x	FIT
5.8816%	8.3399%	6.0102%

Table 6 reports the results for the CntToRfm benchmark. We compare TinyOS (both versions 1 and 2) and FIT-E. We did not compare for Mantis OS because it currently lacks implementation for this benchmark. TinyOS-2.x has larger overhead than TinyOS-1.x in terms of CPU utilization because TinyOS-2.x computes the lowest low power mode to enter into before the system becomes idle, which consumes extra CPU cycles. As expected, FIT's overhead is larger than TinyOS-1.x (as two levels of scheduling are involved). FIT's overhead is lower than TinyOS-2.x because FIT's scheduling overhead is small compared to the system service (i.e., low power computation) TinyOS-2.x invokes.

10 Conclusion and Future Work

In this paper, we present a novel scheduling system FIT for microsensor platforms. FIT is flexible in terms of supporting customizable application-specific scheduling policies. This is achieved through the careful design of its two-tier hierarchical scheduling framework in FIT. It is lightweight by exploiting the proposed MTSP exploration algorithm that effectively reduces the running thread number to reduce preemptions and memory consumption while ensuring system schedulability. In addition, FIT provides detailed real-time schedulability analysis to predict the real-time behavior of the underlying system running on top of it, thus, helps designers to check if application's temporal requirements can be met in design time.

While we have shown that FIT is a promising scheduling system for implementing complex real-time applications in sensor networks, there are several enhancements and optimizations we would like to explore. In particular, we are currently designing a new language to support programming easily under FIT's system model.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000 and in part by an NSERC Discovery Grant under grant No. 341823-07, NSERC Strategic Grant STPGP 364910-08, FQRNT 2010-NC-131844.

References

[1] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A Macroscope in the Redwoods," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys)*, 2005.

[2] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J.A. Stankovic, T.F. Abdelzaher, J. Hui, and B. Krogh, "VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance," *ACM Trans. Sensor Networks*, vol. 2, no. 1, pp. 1-38, 2006.

[3] M. Li and Y. Liu, "Rendered Path: Range-Free Localization in Anisotropic Sensor Networks with Holes," *Proc. ACM MobiCom*, 2007.

[4] M. Li, Y. Liu, and L. Chen, "Non-Threshold Based Event Detection for 3d Environment Monitoring in Sensor Networks," *IEEE Trans. Knowledge and Data Eng.*, vol. 20, no. 12, pp. 1699-1711, Dec. 2008.

[5] M. Li and Y. Liu, "Underground Coal Mine Monitoring with Wireless Sensor Networks," *ACM Trans. Sensor Networks*, 2009.

[6] X. Liu, Q. Wang, L. Sha, and W. He, "Optimal QoS Sampling Frequency Assignment for Real-Time Wireless Sensor Networks," *Proc. IEEE Real-Time Systems Symp. (RTSS)*, 2003.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.E. Culler, and K.S.J. Pister, "System Architecture Directions for Networked Sensors," *Proc. ACM Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[8] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a Lightweight and Flexible Operating System for Tiny Networked Sensors," *Proc. IEEE Workshop Embedded Networked Sensors (EmNets)*, 2004.

[9] C.-C. Han, R. Kumar, R. Shea, and E. Kohler, M. Srivastava, "A Dynamic Operating System for Sensor Nodes," *Proc. ACM Int'l Conf. Mobile Systems, Applications, and Services (MobiSys)*, 2005.

[10] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks and Applications J.*, special issue on wireless sensor networks, vol. 10, pp. 563-579, 2005.

[11] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz, "T2: A Second Generation OS for Embedded Sensor Networks," Technical Report TKN-05-007, Telecomm. Networks Group, Technical Univ. Berlin, 2005.

[12] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks," *Proc. IEEE Real-Time Systems Symp. (RTSS)*, 2005.

[13] H. Cha, S. Choi, I. Jung, H. Kim, and H. Shin, "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks," *Proc. ACM/IEEE Int'l Conf. Information Processing in Sensor Networks (IPSN)*, 2007.

[14] Q. Cao, T.F. Adbelzaher, and J.A. Stankovic, "The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks," *Proc. ACM/IEEE Int'l Conf. Information Processing in Sensor Networks (IPSN)*, 2008.

[15] W. Dong, C. Chen, X. Liu, K. Zheng, R. Chu, and J. Bu, "FIT: A Flexible, LIght-Weight, and Real-Time Scheduling System for Wireless Sensor Platforms," *Proc. IEEE/ACM Int'l Conf. Distributed Computing in Sensor Systems (DCOSS)*, 2008.

[16] G. Lipari, J. Carpenter, and S. Baruah, "A Framework for Achieving Inter-Application Isolation in Multiprogrammed Hard Real-Time Environments," *Proc. IEEE Real-Time Systems Symp. (RTSS)*, 2000.

[17] J. Regehr and J.A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," *Proc. IEEE Real-Time Systems Symp. (RTSS)*, 2001.

[18] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving Real-Time Systems Using Hierarchical Scheduling and Concurrency Analysis," *Proc. IEEE Real-Time Systems Symp. (RTSS)*, 2003.

[19] E. Trumpler and R. Han, "A Systematic Framework for Evolving TinyOS," *Proc. IEEE Workshop Embedded Networked Sensors (EmNets)*, 2006.

[20] W.P. McCartney and N. Sridhar, "Abstractions for Safe Concurrent Programming in Networked Embedded Systems," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys)*, 2006.

[21] C. Duffy, U. Roedig, J. Herbert, and C.J. Sreenan, "Adding Preemption to TinyOS," *Proc. Workshop Embedded Networked Sensors (EmNets)*, 2007.

[22] J.W.S. Liu, *Real-Time Systems.* Prentice Hall, 2000.

[23] T.-W. Kuo, Y.-H. Liu, and K.-J. Lini, "Efficient On-Line Schedulability Tests for Priority Driven Real-Time Systems," *Proc. IEEE Real Time Technology and Applications Symp. (RTAS)*, 2000.

[24] D.I. Katcher, S.S. Sathaye, and J.K. Strosnider, "Fixed Priority Scheduling with Limited Priority Levels," *IEEE Trans. Computers*, vol. 44, no. 9, pp. 1140-1144, Sept. 1995.

[25] R. Cayssials, J. Orozco, J. Santos, and R. Santos, "Rate Monotonic Scheduling of Real-Time Control Systems with the Minimum Number of Priority Levels," *Proc. IEEE Euromicro Conf. Real-Time Systems (ECRTS)*, 1999.

[26] Y. Wang and M. Saksena, "Scheduling Fixed Priority Tasks with Preemption Threshold," *Proc. IEEE Conf. Real-Time Computing Systems and Applications (RTCSA),* 1999.

[27] P. Puschner and A. Burns, "A Review of Worst-Case Execution-Time Analysis," *J. Real-Time Systems,* vol. 18, nos. 2/3, pp. 115-128, May 2000.

[28] S. Mohan, F. Mueller, D. Whalley, and C. Healy, "Timing Analysis for Sensor Network Nodes of the Atmega Processor Family," *Proc. IEEE Real Time and Embedded Technology and Applications Symp. (RTAS),* 2005.

[29] T. Adbelzaher, V. Sharma, and C. Lu, "A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling," *IEEE Trans. Computers,* vol. 53, no. 3, pp. 334-350, Mar. 2004.

[30] T. Abdelzaher, G. Thaker, and P. Lardieri, "A Feasible Region for Meeting Aperiodic End-to-End Deadlines in Resource Pipelines," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS),* 2004.

[31] X. Liu and T. Abdelzaher, "On Non-Utilization Bounds for Arbitrary Fixed Priority Policies," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* 2006.

[32] L.C. DiPippo, V.F. Wolfe, L. Esibov, G. Cooper, R. Bethmangalkar, R. Johnston, B.M. Thuraisingham, and J. Mauer, "Scheduling and Priority Mapping for Static Real-Time Middleware," *Real-Time Systems,* vol. 20, no. 2, pp. 155-182, 2001.

[33] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM Conf. Programming Language Design and Implementation (PLDI),* 2003.

[34] B.L. Titzer, D.K. Lee, and J. Palsberg, "Avrora: Scalable Sensor Network Simulation with Precise Timing," *Proc. ACM/IEEE Int'l Conf. Information Processing in Sensor Networks (IPSN),* 2005.

[35] L. Gu and J.A. Stankovic, "t-kernel: Providing Reliable OS Support to Wireless Sensor Networks," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2006.

Wei Dong received the BS degree from the College of Computer Science at Zhejiang University and achieved all credits in the Advanced Class of Engineering Education (ACEE) of Chu Kechen Honors College from Zhejiang University in 2005. He is currently working toward the PhD degree in the College of Computer Science of Zhejiang University, under the supervision of Prof. Chun Chen. His research interests include networked embedded systems and wireless sensor networks. He is a student member of the IEEE.

Chun Chen received the bachelor degree in mathematics from Xiamen University, China, in 1981, and the masters and PhD degrees in computer science from Zhejiang University, China, in 1984 and 1990, respectively. He is a professor in the College of Computer Science, the dean of the College of Software, and the director of the Institute of Computer Software at Zhejiang University. His research activity is in image processing, computer vision, CAD/CAM, CSCW, and embedded system.

Xue Liu received the BS degree in applied mathematics and the MEng degree in control theory and applications from Tsinghua University and the PhD degree in computer science from the University of Illinois, Urbana-Champaign, in 2006. He is currently an assistant professor in the School of Computer Science, McGill University. He was briefly with the Hewlett-Packard Laboratories (HP Labs) and IBM T.J. Watson Research Center. His research interests include real-time and embedded computing, performance and power management of server systems, sensor networks, fault tolerance, and control. He has filed five patents and published more than 50 research papers in international journals and major peer-reviewed conference proceedings. He is a member of the IEEE and the ACM.

Kougen Zheng received the BSc degree from the North-East Heavy Machinery Institute in 1986 and the PhD degree from the University of Warwick, United Kingdom, in 1992. He is currently a professor in the Computer Science College at Zhejiang University. His research interests include operating systems, computer networks, and artificial intelligence.

Rui Chu received the BS degree in computer science from the National University of Defense Technology in 2001 and the MS and PhD degrees in computer science from the National Laboratory for Parallel and Distributed Processing, in 2003 and 2008, respectively. He is currently an assistant professor in the National Laboratory for Parallel and Distributed Processing. His research interests include parallel and distributed computing, embedded systems, and so on. He is a member of the ACM.

Jiajun Bu received the BS and PhD degrees in computer science from Zhejiang University, China, in 1995 and 2000, respectively. He is currently a professor in the College of Computer Science and the deputy dean of the Department of Digital Media and Network Technology at Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE.