

# FIT: A Flexible, Light-Weight, and Real-Time Scheduling System for Wireless Sensor Platforms

Wei Dong<sup>1</sup>, Chun Chen<sup>1</sup>, Xue Liu<sup>2</sup>, Kougen Zheng<sup>1</sup>, Rui Chu<sup>3</sup>, and Jiajun Bu<sup>1</sup>

973 WSN Joint Lab

<sup>1</sup> College of Computer Science, Zhejiang University

<sup>2</sup> School of Computer Science, McGill University

<sup>3</sup> National University of Defense Technology

dongw@zju.edu.cn, chenc@zju.edu.cn, xueliu@cs.mcgill.ca,  
zkg@zju.edu.cn, rchu@nudt.edu.cn, bjj@zju.edu.cn

**Abstract.** We propose FIT, a flexible, light-weight 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 **light-weight** in terms of minimizing 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 light-weight 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.

## 1 Introduction

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

WSNs typically consist of a large number of micro sensor nodes that self-organize into a multi-hop wireless network. As the complexities for real-world applications 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 has resulted in the need for sophisticated system software. As a large portion of WSN applications are real-time in nature, a good real-time scheduling system plays a central role in task processing in sensor nodes.

The first emerged sensor net OS, TinyOS [3], 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

complicated tasks as task priority preemption is not 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 sensor network OSes, Contiki [4] and SOS [5], both fall into the same category.

As an alternative solution, Mantis OS [6] uses time-sliced multi-threaded scheduling. It supports task preemption and blocking I/O, enabling micro sensor 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 multi-threaded scheduling incurs a higher scheduling overhead because of preemptions and extra memory consumption for thread management.

In this paper, we present a novel scheduling system, FIT, for micro sensor 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 light-weight compared with Mantis OS, as it minimizes the thread number to reduce preemptions and 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.

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 sensor network OSes can be easily incorporated into FIT's two-tier scheduling hierarchy (Section 6.1). It is light-weight as it effectively reduces the running thread number by using MTSP exploration algorithm (Section 6.2). It is real-time as the schedulability analysis is conducted in the MTSP exploration algorithm, thus real-time guarantee can be achieved by employing the *explored* MTSP (Section 6.3).

The rest of this paper is organized as follows: Section 2 describes related work most pertinent to this paper. Section 3 presents our flexible two-tier hierarchical scheduling framework within FIT. Section 4 details the light-weight MTSP exploration algorithm which relies on the real-time schedulability analysis presented in Section 5. Section 6 shows the evaluation results. Finally, we conclude this paper and give future directions of work in Section 7.

## 2 Related Work

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

**Hierarchical scheduling.** Hierarchical scheduling techniques have been used in a number of research projects to create flexible real-time systems, such as PShED [7], HLS [8], etc. While their work focuses on general-purpose PC in open environment, our current work focuses on closed, static, deeply embedded sensor nodes. Regehr *et al.* [9]

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.

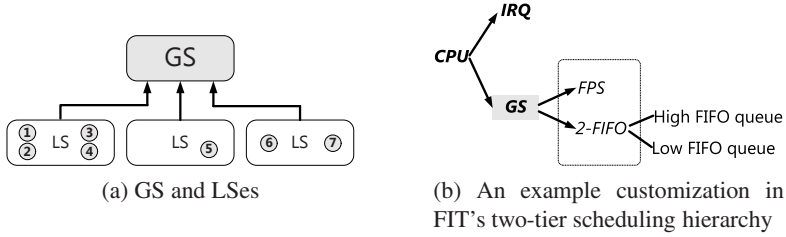
**Task scheduling on sensor nodes.** The event-driven scheme is commonly used in sensornet Oses. TinyOS [3], SOS [5] both fall into this category. In TinyOS, tasks are scheduled in a FIFO manner with a run-to-completion semantics. Like TinyOS, the SOS [5] scheduling policy is also non-preemptive and hence can not provide good real-time guarantee. Mantis OS [6] uses a different scheme. It supports time-sliced preemptive multi-threading. Similar to Mantis OS, Nano-RK [10] provides multi-threading support and uses priority-based preemptive scheduling. It performs static schedulability analysis to provide real-time guarantee. A large body of work was also devoted to improving the capability or the efficiency of task scheduling based on TinyOS [9,11,12,13].

FIT differs from most existing work in sensornet Oses in many important ways. Its two-tier hierarchical framework supports customizable application-specific scheduling policies, hence FIT is more flexible. FIT is also *light-weight* in terms of effectively reducing the running thread number while at the same time ensuring schedulability. Compared with [10], we adopt an event-based programming style, hence it has the opportunity to effectively reduce the running thread number while ensuring schedulability. Compared with [13], FIT automatically assign tasks to appropriate scheduling queues according to their temporal requirements, and provides detailed schedulability analysis.

**Real-time scheduling.** There is a large body of work devoted to real-time scheduling [14]. Priority mapping and PTS (Preemption Threshold Scheduling) are the most pertinent ones to our work. Our current work is different from traditional priority mapping [15] in that each task has a two-dimensional priority. Within the same global priority, we differentiate tasks by their local priorities. Hence, with the same number of global priorities (preemption levels), there are possibilities that MTSP exploration algorithm generates feasible assignment that those approaches cannot. We not only try to overlap the global priorities (preemption levels) but also try to overlap the local priorities. Our work is also different from PTS [16] as it has no notion of preemption threshold, thus priorities do not change at run-time. Finally, our work is different from the abovementioned ones since tasks in FIT consist of multiple jobs with run-to-completion semantics. We analyze schedulability for tasks instead of individual jobs. So the schedulability analysis is different from existing ones. Besides, we also take into account resource access time, hence the analysis is better suited to realistic situations.

### 3 Flexible Two-Tier Hierarchical Framework

Scheduling policies in current sensornet Oses 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



**Fig. 1.** Two-tier scheduling hierarchy

overhead to application programmers. Another fact in current sensornet OSe 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.

### 3.1 Two-Tier Scheduling Hierarchy

Our two-tier scheduling hierarchy as depicted in Figure (a) comprises of two tiers of schedulers. The first-tier scheduler, GS, is designed to schedule LSes. The global scheduling policy employed by the GS is *preemptive* priority scheduling. The GS schedules LSes according to their *global priorities*. The second-tier schedulers, LSes, 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 non-preemptive manner. 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 [5]. 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 LSes and each LS's scheduling policy are customized for different applications.

Figure (b) illustrates an example customization in our two-tier scheduling hierarchy. In this example, our GS schedules two LSes (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 non-preemptively 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.

### 3.2 System Model

We formally define the system model and notations which will be used in the rest the paper in this subsection. As discussed in Section 3.1, in FIT system, each task is assigned a global priority and local priority. A task with higher global priority can preempt a task with lower global priority. Tasks with the same global priority are scheduled by the same LS within which they may have different local priorities. LS schedules tasks in a non-preemptive manner. Tasks with both the same global priority and local priority are scheduled in a FIFO manner.

Whenever we say task A has a higher priority than task B, we mean 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. Task A and task B have the same priority if and only if they have the same global priority and the same local priority.

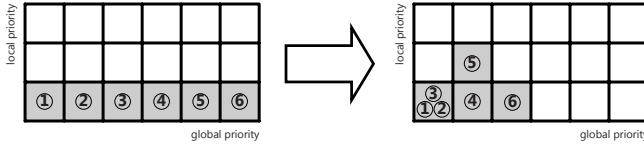
A task comprises of several jobs written with a run-to-completion semantics, i.e., they cannot suspend themselves. The basic scheduling unit of our system is a job. Because I/Os must be done in split phases, we assume *request* to be done at the end of a job while *signal* invokes the start of the next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

We formally define the system model as follows: The system,  $\Gamma$ , consists of a set of  $n$  tasks  $\tau_1, \dots, \tau_n$ . Each task  $\tau_i$  is activated by a periodic sequence of events with period  $T_i$  and is specified a deadline  $D_i$ . A task,  $\tau_i$ , 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,  $B_{ij}$  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.

We make the following assumptions: (1) Task deadline is specified no longer than task period, i.e.,  $D_i \leq T_i$ . (2) A good estimation of  $C_{ij}$  and  $b_{ij}$  is available. (3) All jobs within a task share the common period (i.e., jobs within a task are periodic albeit written with a run-to-completion semantics), and, have the same global priority and local priority, i.e.,  $G_{ij} = G_i \wedge L_{ij} = L_i, \forall 1 \leq j \leq |\tau_i|$ .

## 4 Light-Weight MTSP Exploration

In the previous section, we discussed the flexible two-tier hierarchical scheduling framework 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 schedulability. It is worth noting that we make a distinction between thread and task. Task is from the perspective of functionality while



**Fig. 2.** Global priority and local priority assignment

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 OSES, 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.

#### 4.1 Problem Formulation

Because the number of global priorities maps directly to the number of LSEs (which are implemented as threads), to find an MTSP, we can 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 Figure 2). This can be seen as the most capable scheduling policy as DM is optimal in FPS [14] conforming the assumptions under our system model. We call the initially assigned priority *natural priority*. Then we try to map the natural priorities to the one with as few global priorities as possible, thus to reduce the running thread number. Hence, memory consumption can be reduced, which is important for resource-constrained sensor nodes.

So the problem is that given a task set  $\Gamma = \{\tau_1, \dots, \tau_n\}$  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

$$\text{minimize } |G|, \text{ where } G = \{g_i\} \quad \text{subject to } \text{all tasks are schedulable.} \quad (1)$$

We take *direct mapping* as the mapping rule as in [15] and [17].

**Definition 1 (direct mapping).** Assignment ①  $\mapsto$  Assignment ② is a direct mapping, then, if any task  $\tau_i$  has higher priority than any task  $\tau_j$  in Assignment ①, task  $\tau_j$  cannot have higher priority than that of task  $\tau_i$  in Assignment ②.

As an illustrative example, let's look at the assignment in Figure 2. The left part assignment gives a separate global priority to each task. It implies that up to 6 LSEs are needed which corresponds to 6 threads. This is a relatively costly scheme. It may be fine in traditional general-purpose OSES where CPU speed and memory capacity are abundant. However, for sensor platforms where resources are usually limited, the left assignment is not desirable. Using 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 3.

**Algorithm 1:** MTSP exploration algorithm

---

**Input:** A task set  $\Gamma$  with decreasing deadlines  
**Output:** Assignment to  $(G_i, L_i) = (g_i, l_i)$

<pre> 1: for <math>i \leftarrow 1, n</math> do 2:   <math>(G_i, L_i) \leftarrow (i, 0)</math> 3: end for 4: if test_all_task() != TRUE then 5:   print unschedulable task set 6:   return 7: end if 8: for <math>i \leftarrow 2, n</math> do 9:   save <math>G_i</math> and <math>L_i</math> </pre>	<pre> 10:  <math>(G_i, L_i) \leftarrow (G_{i-1}, L_{i-1})</math> 11:  if test_task(<math>i</math>) == TRUE then 12:    continue 13:  end if 14:  <math>L_i = L_i + 1</math> 15:  if test_task(<math>i</math>) == TRUE then 16:    continue 17:  end if 18:  restore <math>G_i</math> and <math>L_i</math> 19: end for </pre>
---	--

---

## 4.2 MTSP Exploration Algorithm

Our MTSP exploration algorithm is presented in Algorithm 1. After initially assigning 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 MTSP exploration algorithm is  $O(n)$ .

Again, let's revisit Figure 2 as an example. First, we start with each task assigned a 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 non-schedulable 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  $\tau_2$ . We test whether it is schedulable when assigned with the same global priority and local priority as the previous one ( $\tau_1$  in this case). If it is, iterate for the next one ( $\tau_3$  in this case). Otherwise, we test whether  $\tau_2$  is schedulable when assigned with a higher local priority and with the same global priority as the last one  $\tau_1$ . If it is, iterate for the next task. If  $\tau_2$  is not schedulable 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, notice we only test the schedulability of  $\tau_i$  in the loop. This is ensured by the following theorem<sup>1</sup>.

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

## 5 Real-Time Schedulability Analysis

In this section, we consider the real-time constraints presented in the minimization problem in Section 4.1. It is worth mention that our schedulability analysis is conducted on

<sup>1</sup> The proof of Theorem 1 and the *optimality* of MTSP are given in [18] due to space limit.

tasks instead of individual jobs. To derive the schedulability test, We analyze the processor demand [14] of each task. In FIT, the processor demand of a task consists of (1)  $C_i$ : the execution time of  $\tau_i$ . (2)  $I_{lp}$ : interference time from lower priority tasks. (3)  $I_{sp}$ : interference time from the same priority tasks. (4)  $I_{hp}$ : interference time from higher priority tasks. (5)  $B_i$ : blocking time to access shared resources.

A sufficient condition to make  $\tau_i$  schedulable is [14]:  $\min_{0 < t \leq D_i} \frac{W_i(t)}{t} \leq 1$ , where  $W_i(t) = C_i + I_{lp} + I_{sp} + I_{hp} + B_i$ .

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.

**Determining  $C_i$ .**  $C_i$  is  $\tau_i$ 's execution time which equals to the sum of the execution time of all jobs within  $\tau_i$ , i.e.,  $C_i = \sum_{j=1}^{|\tau_i|} C_{ij}$ .

**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  $\tau_i$ , they can never block the execution of  $\tau_i$ . Thus they introduce no interference. Tasks with the same global priority but with lower local priorities, however, can block the execution of  $\tau_i$  as they are scheduled non-preemptively with  $\tau_i$ . 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 interference.

The maximum interference occurs when there is a lower priority task executing each time a job in  $\tau_i$  releases.  $\tau_i$  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,

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

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

$$I_{lp} \leq \underbrace{C_{k_1j_1} \left\lceil \frac{t}{T_{k_1}} \right\rceil + C_{k_2j_2} \left\lceil \frac{t}{T_{k_2}} \right\rceil + \dots + C_{k_mj_m} + \dots + C_{k_mj_m}}_{\text{There are total } |\tau_i| \text{ number of } C_{kj}}. \quad (2)$$

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

**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  $\tau_i$ . We denote  $sp(i)$  as the task set in which tasks have the same priority as  $\tau_i$ .

In FIT, tasks in  $sp(i)$  are scheduled in a FIFO manner with  $\tau_i$ . As with the computation of  $I_{lp}$ ,  $\tau_i$  can be blocked at most  $|\tau_i|$  times. The blocking time each time will not exceed *all* jobs ahead of  $\tau_i$  in the same priority FIFO. As there is only one active job

among a task, the blocking time each time is at most  $\sum_k \max_j \{C_{kj}\}$ , where  $k \in sp(i)$ ,  $1 \leq j \leq |\tau_k|$ . Hence,

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

As with the same reasoning in Section 5, a job,  $J_{kj}$ , may not block  $\tau_i$  for  $|\tau_i|$  times, because it is further limited by  $\lceil t/T_k \rceil$ . For each  $k \in sp(i)$ ,  $1 \leq j \leq |\tau_k|$ , we sort  $\{C_{kj}\}$  in non-increasing order, i.e.,  $C_{kj_1} \geq C_{kj_2} \geq \dots \geq C_{kj_{|\tau_k|}}$ . The interference caused by  $\tau_k$  is

$$I_k \leq \underbrace{C_{kj_1} \left\lceil \frac{t}{T_k} \right\rceil + C_{kj_2} \left\lceil \frac{t}{T_k} \right\rceil + \dots + C_{kj_m} + \dots + C_{kj_m}}_{\text{There are total } |\tau_i| \text{ number of } C_{kj}}. \quad (3)$$

Also, we end up with either  $|\tau_i|$  number of  $C_{kj}$  are added up or we have added all the terms under consideration. Then we consider the (worst-case) overall interference, i.e.,  $I_{sp} \leq \sum_{k \in sp(i)} I_k$ .

**Determining  $I_{hp}$ .**  $I_{hp}$  is the maximum interference caused by higher priority tasks. As all higher priority tasks can interfere the execution of  $\tau_i$ , we denote  $hp(i)$  as the task set in which tasks have higher priority than  $\tau_i$ . The processor demand of higher priority tasks is only limited by  $I_{hp} \leq \sum_{k \in hp(i)} C_k \lceil t/T_k \rceil$ , where  $C_k = \sum_{j=1}^{|\tau_k|} C_{kj}$ .

**Determining  $B_i$ .**  $\tau_i$  blocks  $|\tau_i| - 1$  times (assume there is no blocking for the first job to execute) to access shared resources. As discussed in Section 3.2,  $B_{ij}$  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,  $B_i = \sum_{j=2}^{|\tau_i|} B_{ij} = \sum_{j=2}^{|\tau_i|} (b_{ij} + b'_{ij})$ .

The resource service time  $b_{ij}$  is estimated beforehand 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  $b'_{ij} \leq \sum_{m \in rc(i, j)} \max_{k \in rc_m(i, j)} \{b_{mk}\}$ .

## 6 Evaluation

We implemented FIT on MICAz nodes and carried out extensive tests to evaluate the flexibility, lightweightness and real-time performance of FIT. First, it examines whether FIT is flexible enough to incorporate scheduling policies in existing sensor network OSes; it also measures the overhead as compared to TinyOS and Mantis OS. Second, by a case study of typical workloads on sensor nodes, it examines whether FIT can effectively reduce the running thread number to reduce preemptions and memory consumption. Finally, it examines the real-time performance of FIT.

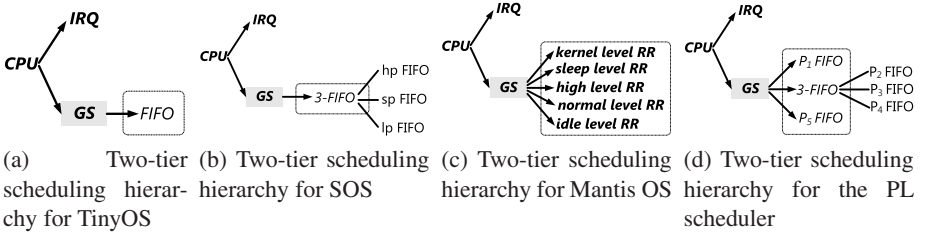


Fig. 3. Existing scheduling policies under FIT's two-tier scheduling hierarchy

## 6.1 Flexibility

A series of existing scheduling policies in sensornet OSEs can be implemented under FIT's two-tier scheduling hierarchy. Figure (a) shows the two-tier scheduling hierarchy for TinyOS [3]. 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 non-preemptive FIFO manner. Figure (b) shows the two-tier scheduling hierarchy for SOS [5]. Also, it has one LS, i.e., the 3-FIFO-LS, which manages three FIFO queues of high, system and low priorities respectively. The scheduling hierarchy of Mantis OS [6] is shown in Figure (c). It has five LSEs with different global priorities. For each LS, different from the previous two, is a *preemptive* round-robin scheduler. Although as we have discussed in Section 3.1, each LS is assumed as a non-preemptive scheduler, FIT's two-tier scheduling hierarchy is general and do allow each LS to employ arbitrary scheduling policy. The real-time schedulability analysis and MTSP exploration algorithm, however, should be revised once this assumption is broken. Figure (d) shows the scheduling hierarchy of the PL scheduler [13] for TinyOS-2.x. The GS schedules three LSEs in a preemptive manner. The 3-FIFO-LS manages three FIFO queues of different local priorities and schedules tasks in a non-preemptive manner. As we can see from Figure 3, FIT's two-tier scheduling hierarchy is very flexible as a series of existing scheduling policies can be easily implemented under this framework.

We evaluate the implementation overhead of FIT's two-tier hierarchical scheduling via four metrics. (1) Overhead of posting a job; (2) Overhead of scheduling consecutive 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; (3) Overhead of LS creation which corresponds to thread creation in Mantis OS; (4) Overhead of scheduling jobs in different LSEs which incurs context switching.

We measure the clock cycles of each operation by using Avrora [19]. The results are reported in Table 1. The post operation of TinyOS is 42cc (cc is a shorthand for clock cycles) while FIT has 90cc. The extra overhead lies in the fact that we make scheduling decisions in our post operation while TinyOS does not need to as priority preemption is not supported. When just the post operation is measured, FIT takes about 44cc which is very close to that of TinyOS. Scheduling jobs in the same LS, however, consumes more time than TinyOS, because FIT will relinquish the CPU control to the GS whenever a job returns. Then the GS passes the CPU control down to the LS which schedules the

**Table 1.** Implementation overhead of FIT’s two-tier hierarchical scheduling compared to TinyOS and Mantis OS (in clock cycles)

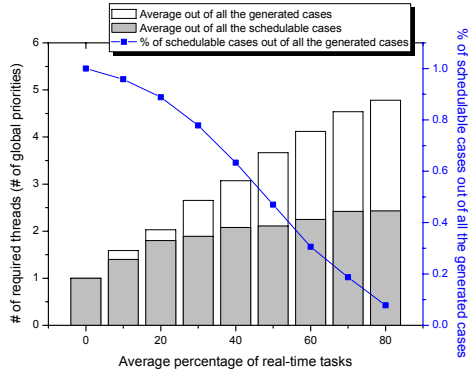
Operations	TinyOS	Mantis OS	FIT
1. Posting a job	42	N/A	90
2. Scheduling jobs in the same LS	63	N/A	130
3. LS/thread creation	N/A	2481	366
4. Scheduling jobs in different LSes	N/A	447	409

next job. In FIT, LS scheduling is about 55cc which is close to TinyOS; GS scheduling consumes about 67cc which is an extra overhead introduced by our two-tier hierarchical scheduling. LS/thread creation in FIT consumes 366cc, as opposed to 2481cc in Mantis OS. The significant difference stems from the fact that Mantis OS uses a dynamic memory allocation which consumes about 1655cc as well as invoking a thread dispatching routine which consumes about 447cc. Without considering these overheads, it leaves 379cc, a little larger than that of FIT. In FIT, scheduling jobs in different LSes involves context switching, which consumes 409cc, 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 a quite acceptable implementation overhead.

## 6.2 Lightweightness

To evaluate the lightweightness of FIT, 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  $C_{ij} \sim U(300, 50)$  and real-time tasks with  $C_{ij} \sim U(10, 8)$ . 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 1–24ms. 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 1000 cases of task sets for each percentage. Note some of the task sets might be schedulable while others are not. The average results are shown in Figure 4. 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% as the majority of the cases are



**Fig. 4.** Number of required threads vs. Average percentage of real-time tasks

unschedulable. To get a closer look at the cases when the task sets are schedulable, as shown by the gray bar in Figure 4, we observe the average number of threads out of all the schedulable cases never exceeds 3, reducing at least 2 threads compared with the worst case.

### 6.3 Real-Time Guarantee

Due to space limit, 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 1ms ( $R_A$ ) and 18ms ( $R_B$ ) respectively. The system consists of the following tasks: (1) Task  $\tau_1$  that consists of 4 jobs. The individual job execution times are selected as 3ms, 4ms, 5ms and 3ms respectively. The shared resources accessed between consecutive jobs are chosen to be  $R_A$ ,  $R_B$  and  $R_A$ . Task  $\tau_1$ ’s period ( $T_1$ ) and deadline ( $D_4$ ) are set to 1s, i.e.,  $T_1 = D_1 = 1s$ . (2) Task  $\tau_2$  with the same setting as  $\tau_1$ . (3) Task  $\tau_3$  with the same setting as  $\tau_1$ . (4) Task  $\tau_4$  that consists of 3 jobs. The individual job execution times are selected as 3ms, 4ms and 18ms 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 20ms to 160ms. (5) Task  $\tau_5$  that consists of 3 jobs. The individual job execution times are selected as 3ms, 36ms and 524ms. The shared resources accessed between consecutive jobs are chosen to be  $R_A$  and  $R_B$ .  $T_5 = D_5 = 10s$ .

Figure (a) and Figure (b) illustrate the cases when we vary the deadline  $D_4$  within 40ms–200ms and 200ms–800ms respectively. We can see from Figure (a) that from the theoretical analysis discussed in Section 5, the system will be schedulable when  $D_4$  reaches 120ms. When  $D_4 < 120ms$ , there could be missed deadlines from the theoretical analysis. The results collected from testbed show that when  $D_4 \geq 100ms$ , employing MTSP (with two LSeS) will result in no missed deadlines in practice. The gap (between 120ms and 100ms) exists because the theoretical analysis considers the *worst* case while in practice the *runtime* interference could be smaller, thus mitigate the percentage of missed deadlines even in the face of an unschedulable system. When  $D_4 < 120ms$ , FIT’s real-time schedulability analysis will also report  $\tau_4$  as the unschedulable task, thus designers can take various ways (e.g., relaxing its deadline, reducing concurrency,

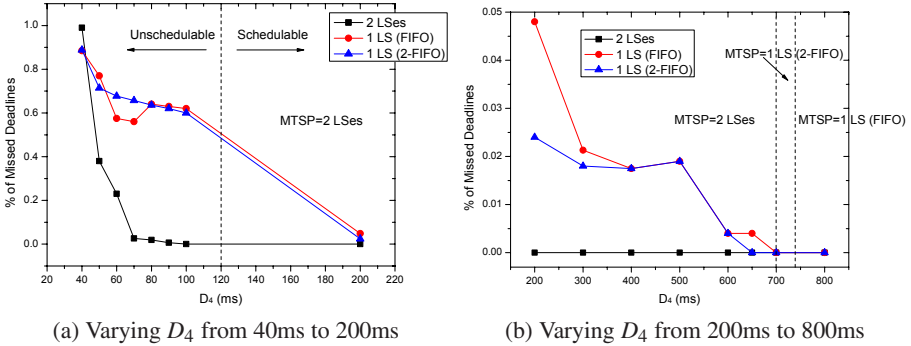


Fig. 5. Varying  $D_4$  from 40ms to 800ms

etc) to redesign a schedulable system. Figure (a) also indicates that using the theoretical analysis, when  $D_4 \geq 120$ ms FIT can ensure that there are no missed deadlines by employing MTSP while the simple FIFO or 2-FIFO scheduling policies may produce missed deadlines in practice. Also notice that the percentage of missed deadlines may increase in Figure (a). This is because as  $D_4 (= T_4)$  increases, the total number of jobs released are reduced.

Figure (b) 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 a FIFO scheme. In practice, this transitional region (700ms–740ms) is located with a smaller  $D_4$  (650ms–700ms). As we can see from Figure (b), in practice, we could employ 2-FIFO in 650ms–700ms as the 2-FIFO scheme will result in no missed deadlines while the FIFO scheme still causes a small fraction of missed deadlines. Anyway, as Figure (a) and Figure (b) indicate, FIT always selects the scheduling policy with minimum overhead while at the same time ensuring the schedulability of the system.

## 7 Conclusion and Future Work

In this paper, we present a novel scheduling system, FIT, for micro sensor 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 light-weight by exploiting the proposed MTSP exploration algorithm which 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.

**Acknowledgements.** The authors would like to thank Prof. Lionel Ni and Prof. Yunhao Liu for their valuable input. 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.

## References

1. Liu, Y., Li, M.: Iso-Map: Energy-Efficient Contour Mapping in Wireless Sensor Networks. In: ICDCS 2007 (2007)
2. Yang, Z., Li, M., Liu, Y.: Sea Depth Measurement with Restricted Floating Sensors. In: RTSS 2007 (2007)
3. TinyOS: <http://www.tinyos.net>
4. Dunkels, A., Grönvall, B., Voigt, T.: Contiki—a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: EmNets 2004 (2004)
5. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A Dynamic Operating System for Sensor Nodes. In: MobiSys 2005 (2005)
6. Bhatti, S., Carlson, J., Dai, H., et al.: MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *MONET Journal, Special Issue on Wireless Sensor Networks* 10, 563–579 (2005)
7. Lipari, G., Carpenter, J., Baruah, S.: A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In: RTSS 2000 (2000)
8. Regehr, J., Stankovic, J.A.: HLS: A framework for composing soft real-time schedulers. In: RTSS 2001 (2001)
9. Regehr, J., Reid, A., Webb, K., Parker, M., Lepreau, J.: Evolving real-time systems using hierarchical scheduling and concurrency analysis. In: RTSS 2003 (2003)
10. Eswaran, A., Rowe, A., Rajkumar, R.: Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In: RTSS 2005 (2005)
11. Trumpler, E., Han, R.: A Systematic framework for evolving TinyOS. In: EmNets 2006 (2006)
12. McCartney, W.P., Sridhar, N.: Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In: SenSys 2006 (2006)
13. Duffy, C., Roedig, U., Herbert, J., Sreenan, C.J.: Adding Preemption to TinyOS. In: EmNets 2007 (2007)
14. Liu, J.W.S.: *Real-Time Systems*. Prentice-Hall, Englewood Cliffs (2000)
15. Sathaye, S.S., Katcher, D.I., Strosnider, J.K.: Fixed Priority Scheduling with Limited Priority Levels. *IEEE Trans. Comput.* 44(9), 1140–1144 (1995)
16. Wang, Y., Saksena, M.: Scheduling fixed priority tasks with preemption threshold (1999)
17. DiPippo, L.C., Wolfe, V.F., et al.: Scheduling and Priority Mapping for Static Real-Time Middleware. *Real-Time Systems* 20(2), 155–182 (2001)
18. Dong, W., Chen, C., Liu, X.: FIT: A Flexible, Lightweight and Realtime Scheduling System for Wireless Sensors. Technical report, Zhejiang University (2007)
19. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable Sensor Network Simulation with Precise Timing. In: IPSN 2005 (2005)