Exploiting Computational Redundancy for Efficient Recovery from Soft Errors in Sensor Nodes

Aly Farahat
Department of Computer Science
Michigan Technological University
Houghton MI 49931, U.S.A.
Email: anfaraha@mtu.edu

Ali Ebnenasir
Department of Computer Science
Michigan Technological University
Houghton MI 49931, U.S.A.
Email: aebnenas@mtu.edu

***Abstract*—Most existing techniques for the design and implementation of fault tolerance use resource redundancy. As such, due to scarcity of resources, it is difficult to directly apply them for adding fault tolerance to sensor nodes in Wireless Sensor Networks (WSNs). Thus, it is desirable to develop techniques that implement fault tolerance under the constraints of memory and processing power of sensor nodes. We present a novel method for designing recovery from transient faults that cause non-deterministic bit-flips in the task queue of the scheduler of TinyOS, which is the operating system of choice for sensor nodes. Specifically, our approach exploits computational redundancy for the design of recovery instead of using resource redundancy. The presented fault-tolerant task queue recovers from bit-flips with significantly lower space/time overhead compared with the Error Correction Codes.**

I. Introduction

Wireless Sensor Networks (WSNs) are increasingly used in mission-critical applications (e.g., body sensor networks, habitat monitoring, flood forecasting, etc.), where they have to be deployed in harsh environments (e.g., volcano, forest, battle field, etc.). On one hand, WSNs must exhibit a high degree of service dependability due to application requirements, and on the other hand, unexpected environmental events, i.e., *faults*, may negatively affect their quality of service. For example, *transient faults* may cause non-deterministic bit-flips in the main memory of sensor nodes (a.k.a. *motes*), thereby perturbing the state of the running program to an arbitrary state in its state space. Since the quality of the service provided by the entire sensor network heavily relies on the dependability of the controlling software of motes, fault tolerance techniques should be applied to improve the dependability of motes. Nonetheless, due to their limited computational (e.g., memory and processing power) and energy resources, it is impractical to apply the traditional fault tolerance methods (e.g., Error Correction Code (ECC) [9]) to motes. This paper proposes a novel method that exploits computational redundancy for the addition of recovery to transient bit-flips in the task queue of TinyOS [14].[1]

Most existing techniques [12], [4], [10], [1], [13] present solutions for the design of fault-tolerant protocols for WSNs rather than focusing on the fault tolerance of individual sensor nodes. For instance, techniques for reliable transmission are mostly based on redundant and/or multi-path retransmission [10]. Several methods exist for (i) designing self-stabilizing WSN communication protocols [1] that ensure a correct synchronization among sensor nodes starting from an arbitrary non-synchronized state, and (ii) providing recovery for data dissemination in WSNs [13]. ECC methods (e.g., Hamming code [9]) often require extensive memory redundancy for storing extra parity bits in code words. Moreover, decoding/coding algorithms in these methods are computationally expensive.

We propose a novel approach that enables space/time-efficient recovery to transient Bit-Flips (BFs) in motes. The proposed approach is based on the detection of the violations of invariance conditions that must always be true and dynamic corrections of such violations. Specifically, we focus on the task queue of the TinyOS as it is one of the most critical components of the kernel of TinyOS and its structure is heavily sensitive to BFs. We first define conditions under which the task queue has a valid structure, called the *structural invariant*. Then, before and after the addition/removal of a task to/from the task queue, we check whether the structural invariant holds. In case of the violation of the invariant, we identify different failure scenarios created due to the occurrence of BFs and systematically correct them, thereby recovering to the structural invariant. The proposed approach enables the detection and correction of multiple BFs in a single-byte variable in a space/time-efficient fashion. Compared with the Hamming Code (HC) [9], our approach needs at least 20% less memory and performs at least twice as fast as HC. The time complexity of our approach is linear in the size of the task queue. We also note that HC cannot correct multiple BFs whereas our approach enables the correction of multiple BFs as long as they occur in the same variable. Furthermore, for some special cases, the proposed approach corrects BFs in multiple variables as well.

Organization. Section II illustrates the structure of the TinyOS task queue. Then, Section III presents our approach for the detection and correction of transient bit flips in the TinyOS task queue. We also demonstrate the superiority of the space/time efficiency of the proposed approach compared with the ECC methods. Section IV makes concluding remarks and outlines future research directions.

[1]TinyOS is the operating system of choice for sensor nodes in WSNs.

II. Structural Invariance of TinyOS Task Queue

In this section, we define what constitutes a valid structure of TinyOS's task queue. In Tiny OS version 2.x, the task queue is a linked list of task identifiers implemented as a statically allocated array of 256 entries (see Figure 1). Figure 2 illustrates the implementation of the task queue in nesC [8], which is a component-based variant of the C programming language used for application development on TinyOS. Each identifier (ID) is an integer between 0 and 255 inclusive. The set of variables of interest are `m_head`, `m_tail` and `m_next[256]`. `m_head` holds the index of the oldest ID in the queue, and `m_tail` holds the ID of the most recent task inserted in the queue. Every value in `m_next` is an ID for the next task to be executed and an index (i.e., pointer) to the successor entry in `m_next`. A distinguished task has the identifier `NO_TASK` $= 255$. `NO_TASK` is the value of `m_next[m_tail]` and it is the successor of all non requesting identifiers. For example, as depicted in Figure 1, a queue state s_1 consists of `m_head`=12, `m_next[12]`=3, `m_next[3]`=255, `m_tail`=3, and $\forall j : (j \neq 12) :$ `m_next[j]`=255.

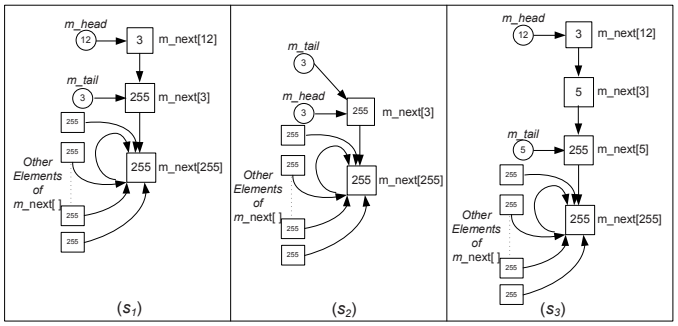

Fig. 1. Example states of the task queue.

The state s_1 represents a task queue having only two pending tasks of identifiers 12 and 3 respectively. The effect of `popTask()` on s_1 is a transition to state s_2 (see Figures 2 and 1). In state s_2, `m_head`=3, `m_next[3]`=255, `m_tail`=3, and $\forall j : (0 \leq j \leq 255) :$ `m_next[j]`=255. The effect of `pushTask(5)` on s_1 is a transition to state s_3 (see Figures 2 and 1), where `m_head`=12, `m_next[12]`=3, `m_next[3]`=5, `m_next[5]`=255, `m_tail`=5, and $\forall j : (j \neq 12) \wedge (j \neq 3) \wedge (0 \leq j \leq 255)$: `m_next[j]`=255.

Structural Invariant. A *valid* state of the task queue is a state where the queue has a linear structure with its head (`m_head`) pointing to its beginning and its tail (`m_tail`) pointing to the most recently added identifier to the task queue. Each element of `m_next` with a non-255 ID is reachable from the head. Each entry of `m_next` that is not in the queue holds the value of `NO_TASK`, and `m_next[m_tail]` is equal to `NO_TASK`. Moreover, the task IDs belong to the interval $0 \leq ID \leq 255$. Figure 3-(a) illustrates a sample valid state of the task queue. Furthermore, any operation performed on the queue should remove an element from the head (i.e., `popTask()`), add an element to the tail `pushTask()` or leave the structure of the queue and the task IDs unchanged.

```
inline uint8_t popTask()
{
 if( m_head != NO_TASK ) {
   uint8_t id = m_head;
   m_head = m_next[m_head];
   if( m_head == NO_TASK )   m_tail = NO_TASK;
   m_next[id] = NO_TASK;
   return id;   }
  else       return NO_TASK;
  }

bool isWaiting( uint8_t id )
 { return (m_next[id] != NO_TASK) || (m_tail == id); }

bool pushTask( uint8_t id )  {
 if( !isWaiting(id) )  {
   if( m_head == NO_TASK ) { m_head = id;  m_tail = id; }
   else  {  m_next[m_tail] = id;  m_tail = id;   }
   return TRUE;
 } else   return FALSE;      }
```

Fig. 2. Excerpt of the Tiny OS Scheduler.

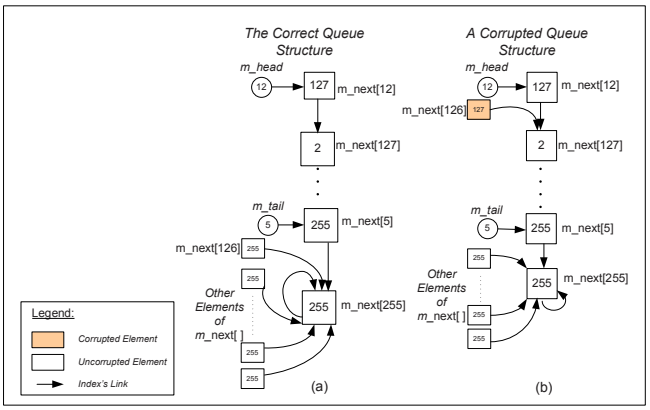

Fig. 3. Valid and invalid task queue structures.

Transient faults. Transient faults may toggle multiple bits in a single variable; i.e., `m_head`, `m_tail` or a memory cell of `m_next[]`. The case of multi-variable corruption is the subject of our current investigation. Bit-flips may perturb a task ID and the structure of the task queue to an invalid state. For example, Figure 3 demonstrates how resetting the most significant bit of `m_next[126]` could change its content from 255 to 127, thereby pointing to `m_next[127]` instead of pointing to `NO_TASK`.

III. Addition of Recovery

Section III-A analyzes the memory and time requirements of correcting BFs with the Hamming code. Section III-B illustrates how our approach enables recovery from BFs by detecting invalid queue structures and correcting them.

A. Correcting Bit-Flips with ECC

One approach for recovery from transient faults that cause bit-flips is to use error detection and correction codes such as the Hamming Code (HC) [9]. However, due to high memory/CPU cost of the encoding/decoding algorithms these approaches seem impractical in the context of WSNs. For example, there are two ways to deal with bit-flips in the task queue using HC; consider either individual memory cells of the `m_next[]` array as separate data words, or the entire 256 bytes of the task queue as one data word.

In the first case, each cell of the `m_next[]` array should be encoded before storing a value and it should be decoded before reading its contents. To encode 8 bits of data with HC, we need 4 extra parity bits, which results in a code word with 12 bits in the following format: $p_1p_2d_1p_3d_2d_3d_4p_4d_5d_6d_7d_8$, where d_j denotes data bits for $1 \leq j \leq 8$, and p_i represents the parity bits for $1 \leq i \leq 4$. The encoding algorithm of HC determines the 12-bit code word by multiplying a 12×8 matrix by a vector made of the data bits. Such a matrix multiplication takes 96 multiplications and 84 additions; i.e., totally 180 *basic operations* in addition to one read and write operation on each memory cell, where a basic operation includes arithmetic and logical operations as well as comparisons and load/store. The decoding algorithm also multiplies a 4×12 matrix by a vector containing the 12-bit code word, which results in a 4-bit syndrome vector representing the position of the corrupted bit. (Thus, each decoding takes 48 multiplications and 44 additions, totally 92 basic operations.) Notice that for each byte allocated in `m_next[]` 4 extra bits should be considered for parity. That is, $256/2 = 128$ extra bytes should be allotted along with the 256 bytes allocated for `m_next[]`. Besides, every time a task ID is stored/retrieved to/from a memory cell in `m_next[]`, the encoding/decoding algorithm must be executed. That is, for one round of detection and correction, $256 \times (180 + 92) = 69632$ basic operations should be performed.

In the second case, the queue comprises a bit pattern with $256 \times 8 = 2024$ bits, for which $1 + log\ 2024 = 12$ parity bits are needed in HC. Thus, the size of the code word is equal to $2024 + 12 = 2036$ bits. The encoding takes $2024 \times 2036 = 4120864$ multiplications and $2023 \times 2036 = 4118828$ additions; i.e., totally 8239692 basic operations. For decoding, we will need 12×2036 multiplications and 12×2035 additions resulting in 48852 basic operations. This analysis clearly illustrates the impracticality of using HC on sensor nodes with a small memory and a limited processing power. While other ECC methods (e.g., , Forward Error Correction (FEC) [3] and Reed-Solomon (RS) [15]) can correct cases where multiple variables are corrupted, they are even more expensive than HC in terms of either space or time. For example, the RS method needs t bits for the correction of $\lfloor t/2 \rfloor$ bits and $\mathcal{O}(t^2)$ is its time complexity.

B. Adding Recovery to Task Queue

This section illustrates how we enable recovery to a valid queue structure from transient BFs. Figure 4 depicts a state machine that demonstrates the impact of transient faults and how recovery should be achieved. Since the task queue is a centralized program running on a single CPU, we can benefit from a high atomicity model in which a set of instructions can be performed atomically. In fact, the nesC language provides atomic blocks that capture a sequence of statements that are supposed to be executed without interruption. The essence of the addition of recovery in high atomicity [6] is based on detecting the violation of the invariant due to the occurrence of faults, and providing recovery from every invalid state to the invariant. Thus, we present the function `DetectCorrect()` (see Figure 5) that we add to the Tiny OS scheduler to enable the detection and correction of BFs before and after any push/pop operations on the task queue. The function `DetectCorrect()` should be invoked in an atomic block (i.e., `atomic{DetectCorrect()}`) to ensure that detection and correction are not interrupted during execution. Depending on the harshness of the environment where the motes are deployed, the period of invoking `DetectCorrect()` could be changed by the developers; i.e., `DetectCorrect()` can be invoked in an adaptive fashion by the scheduler of TinyOS in order to enable a tradeoff between the degree of dependability and the energy cost of providing recovery. Next, we explain different parts of `DetectCorrect()` to illustrate how detection and correction are achieved.

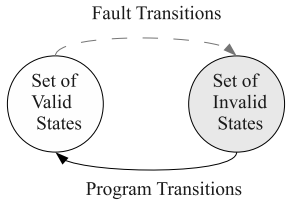

Fig. 4. Adding recovery to the task queue.

Data structures. To detect and correct corruptions of the task queue, `DetectCorrect()` gathers some information about the structure of the queue and stores them in these data structures (see Figure 5):

```
DetectCorrect(int q_size) {
 uint8_t previous;
 uint8_t current=m_head;
 uint8_t Index=0;
 uint8_t dangleElem=0;
 uint8_t non255 =0;      // Number of non-255 elements
 uint8_t qLength =0;     // Number of elements in the queue
 uint8_t cyclePoint =0;  // The corrupted element that
                         // points back and creates a cycle

 component visited = new BitVectorC(256);
 visited.clearAll();
 component pointedTo = new BitVectorC(256);
 pointedTo.clearAll();
 component pointsToNoTask = new BitVectorC(256);
 pointsToNoTask.clearAll();

 bool cyclic = FALSE;
 bool pointedByHead = FALSE;
```

Fig. 5. Data structures.

(1) `previous, current, dangleElem` and `cyclePoint` are pointers that are used during the traversal of the queue; (2) `non255` keeps the number of memory cells in `m_next` that contain non-255 values; (3) `qLength` stores the number of non-255 elements reachable from the head of the queue (`m_head`); (4) the `visited` bit vector allocates one bit corresponding to each element of `m_next` illustrating whether or not it has been visited previously in a queue traversal for cycle detection; (5) the `pointedTo` bit vector keeps a bit for each element of `m_next` demonstrating whether or not that element is being pointed to by some other element; (6) the `pointsToNoTask` bit vector allocates a bit corresponding to each memory cell of `m_next` that

contains NO_TASK; (7) the cyclic flag is set if a cycle is detected in the structure of the task queue, and (8) the pointedByHead flag is true if and only if there is an element in the queue that is pointed by both m_head and another element in the queue. We use the pointedByHead flag in detecting/correcting the corruption of m_head. Notice that, we allocate 96 bytes for the bit vectors and 7 bytes for other variables (i.e., 103 bytes totally) capturing local variables; i.e., when DetectCorrect() returns this memory is released.

Initialization. In this step, we first count the total number of elements in m_next that contain non-255 values. Then, we initialize the pointedTo and pointsToNoTask bit vectors. This step incurs $256 \times 15 = 3840$ basic operations on our solution.

```
// Count the number of non-255 elements in array m_next
 for(Index=0; Index<NO_TASK; ++Index)
    if (m_next[Index] != NO_TASK) non255++;
// Determine the elements that are being pointed to
 for(Index=0; Index<NO_TASK; ++Index)
    pointedTo.set(m_next[Index]);
// Determine the elements that point to NO_TASK
 for(Index=0; Index<NO_TASK; ++Index)
    if(m_next[Index] == NO_TASK) pointsToNoTask.set(Index);
```

Detection and correction of m_head. Since the traversal of the queue for subsequent processing is performed using the m_head pointer, we first ensure that m_head is corrected. If m_head points to NO_TASK (see Figure 6), then we set m_head to the index of the element to which no other element points, and exit (because, by assumption, our focus is on single-variable corruption). Otherwise, we detect whether m_head points to another non-255 element in the queue. (Please see Figure 7 and the first for-loop in the else part of Figure 6.) If so, then we set m_head to the index of the non-255 element to which no other element points. (See the second for-loop in the else part of Figure 6.) Figure 7 illustrates a case where the value of m_head has been corrupted from 12 to 4.

```
if (m_head == NO_TASK)  {
  for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) && m_next[Index] != NO_TASK) {
            m_head = Index;  return; }      }
else {
  for(Index=0; Index<NO_TASK; ++Index)
    if(pointedTo.get(Index) && Index == m_head) {
                      pointedByHead = TRUE; break; }
  if (pointedByHead)
   for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) && m_next[Index] != NO_TASK) {
                    m_head = Index;  return; }
       }
```

Fig. 6. Detect and correct m_head.

The time complexity (and energy consumption) of this step is proportional to the maximum number of basic operations. If m_head = NO_TASK, the for-loop in the if part of Figure 6 will be executed, which has one comparison and one increment for the loop counter in each iteration. Moreover, the if-statement inside the for-loop performs two load operations, one comparison and two logical operations per iteration. Thus, in the worst case, we have 7 basic operations in each iteration of this for-loop, which results in $256 \times 7 = 1792$ basic operations if m_head = NO_TASK. A similar reasoning illustrates that, in the worst case, we perform $256 \times 15 = 3840$ basic operations if m_head $\neq$ NO_TASK. Therefore, since either the if part or the else part is executed in Figure 6, the correction of m_head takes at most 3840 basic operations.

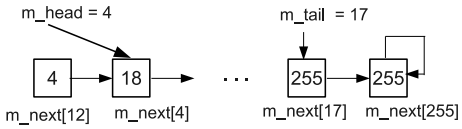

Fig. 7. Corruption of m_head.

Detection and correction of cyclic structures. The do-while loop in the below code uses the visited bit pattern to determine whether there is a cycle in the queue. This loop also stores the number of elements in the queue that are reachable from m_head in the qLength variable.

```
// Detect cycles
do {
   if(!visited.get(current))       visited.set(current);
   else { cyclic = TRUE;
          cyclePoint = previous;     break;  }
   previous=current;
   current=m_next[previous];
   qLength++;
     } while(current != NO_TASK && m_tail!=previous);

// Correct cycles
if (cyclic && (cyclePoint == m_tail)) {
          m_next[cyclePoint] =  NO_TASK; return; }
if (cyclic && cyclePoint != m_tail)
  for(Index=0; Index<NO_TASK; ++Index)
    if(!pointedTo.get(Index) &&
          (m_next[Index] != NO_TASK) && (Index != m_head)) {
          m_next[cyclePoint] = Index;  return; }
```

A cycle could be formed in two ways: either the tail points back to some element including itself (see Figure 8-(a)), or another element points back to some element including itself (see Figure 8-(b)). If a cycle is detected, then the cyclic flag is set and the index of the element pointing back is stored in cyclePoint. In case cyclePoint is equal to m_tail, then that means the tail of the queue is pointing back to some element instead of pointing to NO_TASK. Otherwise, to fix the cycle, we set the contents of m_next[cyclePoint] to the index of the element that has become *dangled* due to the cycle creation; i.e., the element to which no element points, does not point to NO_TASK, and is not equal to m_head. This correction will take at most $256 \times 26 = 6656$ basic operations.

Detection and correction of queue size and non-255 elements. To detect discrepancies in the size of the task queue, we add a new variable q_size to the TinyOS scheduler to store the size of the queue outside the DetectCorrect function. Nonetheless, q_size could be perturbed by transient faults. The first if statement in Figure 9 corrects q_size. Notice that DetectCorrect can simultaneously correct q_size and m_head, which is a special case of correcting MBFs in multiple variables. Moreover, faults may change the value of an array element from 255 to some other value. This means that that element points to some queue element. Such a link is not part of the task queue and should be eliminated.

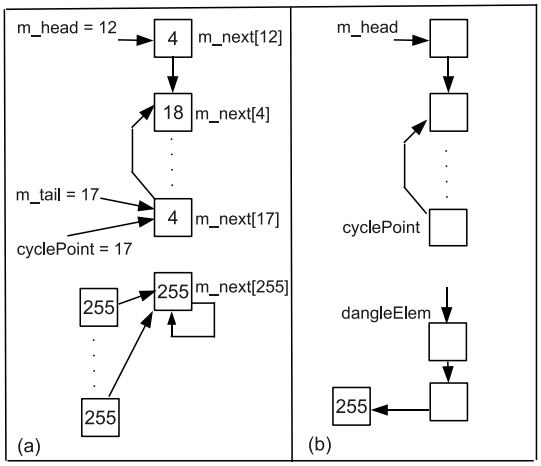

Fig. 8. Cyclic corruption of the task queue.

To this end, we assign 255 to an element to which no other element points, points to a non-255 element and is not equal to `m_head`. The `for-loop` in the second `if statement` could take at most $256 \times 11 = 2816$ basic operations.

```
if (qLength == non255) {
    if (q_size != qLength)  { q_size = qLength; return; }
        else return;  // Task queue is NOT corrupted.
         }

if (non255 > q_size) // some 255 element has become non255
   for(Index=0; Index<NO_TASK; ++Index)
     if (!pointedTo.get(Index) &&
        (m_next[Index] != NO_TASK) && (Index != m_head)) {
                  m_next[Index] = NO_TASK;  return;  }
```

Fig. 9. Detect and correct queue size.

Detection and correction of `m_tail`. To detect and correct the corruptions of `m_tail`, we set `m_tail` to the index of the first element whose contents point to `NO_TASK` (see below). For example, in Figure 10, `m_tail` is set to 17. The `for-loop` in the below code performs at most 2560 basic operations.

```
for(Index=0; Index<NO_TASK; ++Index) {
    if ((!pointsToNoTask.get(Index)) &&
        (m_next[m_next[Index]] == NO_TASK) &&
                 (m_tail != m_next[Index])) {
                m_tail = m_next[Index];  return;  } }
```

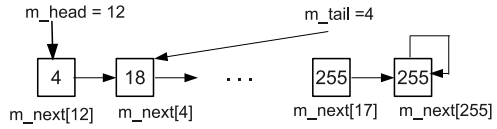

Fig. 10. Corruption of `m_tail`.

Detection and correction of corrupted acyclic structures. If faults corrupt a non-255 element so it points to one of its successors, then a structure similar to Figure 11 could be created. In this example, the contents of `m_next[4]` is changed from 18 to 25 and `m_next[18]` becomes unreachable from head; i.e., a *dangling* element. One way to detect this case is to simply compare `qLength` with the number of non-255 elements; if `qLength` $\neq$ `non255`, then either this case has occurred or the corruption of `m_head`. Nonetheless, if the code of the `DetectCorrect()` routine reaches this point, then it means that `m_head` has the correct value.

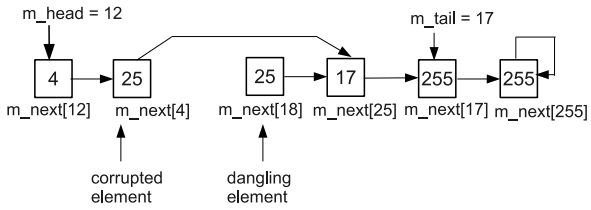

Fig. 11. Corrupted acyclic structure.

The identification of the `corrupted element` in Figure 11 is not straightforward. Our strategy is to determine the index of the element in the queue that is pointed by two internal elements of the queue (see `m_next[25]` in Figure 11). Such an element must be in the fragment of the queue that starts with the dangling element. Thus, we first find the index of the dangling element by the first `for-loop` in Figure 12. If there is such a dangling element, then we reset the `pointedTo` bit vector. Then, in the first `do-while` in Figure 12, we start setting the bits of `pointedTo` corresponding to the fragment of the queue that starts with the dangling element. In the second `do-while`, we search the first fragment of the queue (starting from `m_head`) for the element that points to an element whose corresponding bit is already set in the `pointedTo` vector. Once we find such an element, we set its content to the index of the dangling element, and the queue is corrected. This step includes $27 \times 256 = 6912$ basic operations in the worst case.

```
dangleElem = NO_TASK;
for(Index=0; Index<NO_TASK; ++Index)
   if(!pointedTo.get(Index) &&
       m_next[Index] != NO_TASK &&
       Index != m_head) {
                  dangleElem = Index; break; }

if (dangleElem == NO_TASK)  return;

pointedTo.clearAll();
current = dangleElem;

do {
     pointedTo.set(m_next[current]);
     previous=current;
     current=m_next[previous];
     } while(current != NO_TASK && m_tail!=previous);

current = m_head;

do {
    if (pointedTo.get(m_next[current]) {
              m_next[current] = dangleElem; return; }
    previous=current;
    current=m_next[previous];
     } while(current != NO_TASK && m_tail!=previous);
   }
```

Fig. 12. Detect and correct acyclic structures.

Time complexity of `DetectCorrect()`. Since the code of `DetectCorrect()` does not include nested for-loops, its time complexity is linear in the size of the task queue. Figure 13 presents a comparison of the time/space cost of the proposed method of this paper with two scenarios of using

the Hamming code for correction of BFs: HC1 represents the case where each element of `m_next` is encoded with HC, and HC2 denotes the case where the entire `m_next` is encoded as a single word. Notice that, our approach outperforms HC1 in terms of both time and space efficiency, respectively by a factor of 20% and 60%. More importantly, the required memory (i.e., 103 bytes) is temporary; i.e., when `DetectCorrect()` returns this memory is released. The HC2 method seems impractical due to expensive computing requirements.

Approach	Memory Cost	# of Operations
Hamming Code for each element of `m_next` (HC1)	128 Bytes	$\simeq$ 70000
Hamming Code for the entire `m_next` (HC2)	12 bits	$\simeq$ 4.17 million
Proposed Method	103 Bytes	28000

Fig. 13. Space/Time cost of correction of BFs.

Scope of correction. The scope of correction in these three methods is different. Figure 14 demonstrates that our approach can correct multiple bit-flips in a single variable, which cannot be achieved by HC1 and HC2. However, HC1 can correct single bit-flips in multiple variables, which we do not currently have a solution for it.

Approach	Corrects SBFs	Corrects MBFs in a Variable	Corrects SBFs in Multi Vars	Corrects MBFs in Multi Vars
HC1	Yes	No	Yes	No
HC2	Yes	No	No	No
Proposed Method	Yes	Yes	No	No

Fig. 14. Scope of correction for Single Bit-Flips (SBFs) and Multiple Bit-Flips (MBFs).

Fault tolerance of `DetectCorrect()`. In case transient faults perturb the local variables and/or the control flow of `DetectCorrect()`, the current round of execution of `DetectCorrect()` may not recover the structure of the task queue. However, since `DetectCorrect()` is executed repeatedly and transient faults eventually stop occurring, `DetectCorrect()` will eventually provide recovery.

IV. Conclusions and Future Work

We presented a novel method for the detection and correction of transient Bit-Flip (BF) in the task queue of the TinyOS, which is the operating system of choice for sensor nodes. Since motes have limited computational and energy resources, instead of using resource redundancy, the proposed approach exploits computational redundancy to efficiently recover from transient BFs that corrupt the contents and the structure of the task queue. The essence of our approach is based on the detection of invalid structures of the queue that might be created due to transient faults. Upon reaching an invalid structure, we analyze the structure of the task queue to determine which failure scenario has occurred and recover to a valid state. Using this method, we can correct Multiple BFs (MBFs) in single-byte variables. We illustrate that the proposed approach can provide a better time/space efficiency with respect to Error Correction Codes such as the Hamming code [9] (see Figures 13 and 14).

Several techniques exist for designing fault-tolerant data structures. Aumann *et al.* [2] present alternative implementations for pointer-based data structures by adding redundant links. Finocchi *et al.* [7] provide resilient search trees through periodic checkpoints. Jørgensen *et al.* [11] devise a method that ensures the resilience of priority queues by storing pointers in resilient memory locations. By contrast, our approach continuously monitors a structural invariance and provides recovery if the invariant is violated.

Future/ongoing work focuses on techniques for the correction of MBFs in multiple variables. Moreover, we would like to leverage our previous work [5] on automated addition of fault tolerance for the addition of recovery to data structures. Specifically, our previous work models a state as a unique valuation of variables of primitive types (e.g., Boolean and integer). Nonetheless, we need to create richer models that capture the topological state of complex data structures. We will also work on models where ECC methods and our approach are used in a hybrid fashion.

References

[1] M. Arumugam and S. Kulkarni. Self-stabilizing deterministic TDMA for sensor networks. *Distributed Computing and Internet Technology*, pages 69–81, 2005.

[2] Y. Aumann and M. Bender. Fault tolerant data structures. In *focs*, page 580. Published by the IEEE Computer Society, 1996.

[3] G. C. Clark and J. B. Cain. *Error-Correction Coding for Digital Communications*. Springer, 1981.

[4] T. Clouqueur, P. Ramanathan, K. Saluja, and K. Wang. Value-fusion versus decision-fusion for fault-tolerance in collaborative target detection in sensor networks. In *Proceedings of Fourth International Conference on Information Fusion*. Citeseer, 2001.

[5] A. Ebnenasir. *Automatic synthesis of fault-tolerance*. PhD thesis, Michigan State University, 2005.

[6] A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *To appear in the proceedigns of 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[7] I. Finocchi, F. Grandoni, and G. Italiano. Resilient search trees. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 547–553. Society for Industrial and Applied Mathematics, 2007.

[8] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, page 11. ACM, 2003.

[9] R. W. Hamming. *Coding and Information Theory*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[10] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185. ACM, 1999.

[11] A. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. *Algorithms and Data Structures*, pages 127–138, 2007.

[12] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. Fault tolerance techniques for wireless ad hoc sensor networks. *sensors*, pages 1491–1496, 2002.

[13] S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal of Distributed Sensor Networks*, 2(1):55–78, 2006.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[15] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.