Prius: Generic Hybrid Trace Compression for Wireless Sensor Networks

Vinaitheerthan Sundaram
Purdue University
West Lafayette, IN, USA
vsundar@purdue.edu

Patrick Eugster
Purdue University
West Lafayette, IN, USA
peugster@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, IN, USA
xyzhang@purdue.edu

Abstract

Several diagnostic tracing techniques (e.g., event, power, and control-flow tracing) have been proposed for run-time debugging and postmortem analysis of wireless sensor networks (WSNs). Traces generated by such techniques can become large, defying the harsh resource constraints of WSNs. Compression is a straightforward candidate to reduce trace sizes, yet is challenged by the same resource constraints. Established trace compression algorithms perform unsatisfactorily under these constraints.

We propose Prius, a novel hybrid (offline/online) trace compression technique that enables application of established trace compression algorithms for WSNs and achieves high compression rates and significant energy savings. We have implemented such hybrid versions of two established compression techniques for TinyOS and evaluated them on various applications. Prius respects the resource constraints of WSNs (5% average program memory overhead) whilst reducing energy consumption on average by 46% and 49% compared to straightforward online adaptations of established compression algorithms and the state-of-the-art trace-specific compression algorithm respectively.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing, Debugging aids*

General Terms

Design, Measurement, Performance, Reliability

Keywords

Compression, Tracing, Sensor Networks

1 Introduction

Wireless Sensor Networks (WSNs) are being increasingly deployed in various scientific as well as industrial domains to understand the micro-behavior of physical phenomena. A few prominent deployments include habitat monitoring [43], volcano monitoring [47], precision agriculture [22], permafrost monitoring [14], and micro-climate monitoring [5].

1.1 Deployment Failures

WSNs are highly susceptible to deployment failures as they are deployed *in situ* in austere environments such as volcanoes [47] or mountains [14]. Unexpected failures have been observed in many deployments despite thorough in-lab testing prior to deployment [6, 22, 43, 47, 18]. Even well-tested protocols have exhibited failures in the field [47, 22].

Consider as an example the PermaSense deployment [14] that monitors permafrost in the Swiss Alps. The deployment experienced severe performance degradation after running for 6 months (March 2009). Extensive resets of nodes, up to 40 resets per node per day, were observed for 3 months [18]. The cause of the bug was a lookup task whose running time increased with the lifetime of the network, which after several months of deployment became large enough to cause node resets. The diagnosis took months and several expensive trips to the mountain top.

1.2 Trace Based Debugging

To cope with deployment failures, several run-time diagnostic tracing techniques have been recently proposed for WSNs that enable postmortem diagnosis [41, 19, 37, 21, 20]. These techniques propose efficient recording of different types of traces such as function call traces [21, 36], control-flow traces [41, 37], event traces [19], and power traces [20]. The traces generated by these techniques can provide insight into the execution at the time of failure and thus aid diagnosis. For the Permafrost deployment failure, the control-flow or function call trace would have shown the continuous execution of the lookup task, hinting to the problem.

1.3 Tracing Overhead

The amount of trace information generated and collected by such approaches for diagnosis, however, increases rapidly with the number of components or events traced in the application [37, 41]. Large traces in the order of KBs put pressure on the storage as well as on the radio and consequently, the energy required to manage them.

Consider the case of the TinyTracer, a control-flow tracer [41, 42]. Figure 1 shows the energy *overhead* of uncompressed tracing of the main component of TinyOS applications as well as all components including the system components such as LEDs, sensor, radio, timer used by those

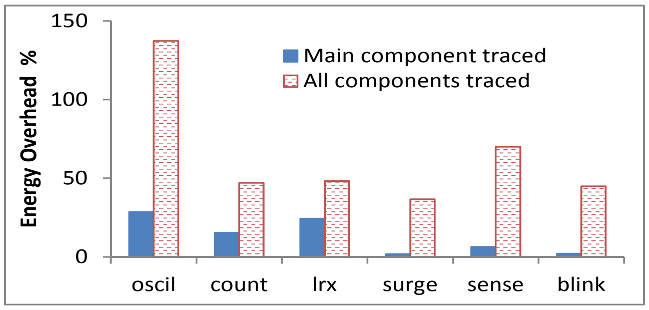


Figure 1. Energy overhead of uncompressed tracing. The y-axis shows energy overhead in % compared to the baseline case, in which no tracing is performed (i.e., energy consumption for the same application without tracing).

applications for a 30 minute run (Section 4.3 describes the benchmark applications in detail). Note that many WSN faults reside in system components or the interactions between the main components and system components, demanding tracing into several components [19, 18, 37, 48, 41]. The energy overhead for uncompressed tracing ranges from 3% (`blink` main component) to 135% (`oscil` all components) of the energy used to run the application for 30 minutes *without tracing*. This represents a significant overhead, which may hamper the feasibility of tracing.

1.4 Trace Compression Challenges

To mitigate the overhead of tracing, a natural approach is to *compress* traces. The extreme resource constraints inherent to WSNs, however, pose novel challenges for compression. Established compression algorithms [35, 46, 29] are either inapplicable or have to be adapted to satisfy the limits on memory and CPU resources. Adaptations of such established algorithms still perform poorly for WSN traces due to inherently small input buffers, which are only a few hundred bytes in WSNs, leading to few opportunities for learning the repeating patterns and replacing them.

The reasons for inherently small input buffers in WSNs are twofold. First, since traces are constantly generated with the execution, they have to be buffered in RAM before compression – otherwise, computation-intensive compression may interfere with trace generation. After compressing a trace buffer, the compressed output has to be buffered in RAM as storage into non-volatile flash or transmission on the radio is slow. Due to the differences in execution speed between trace generation, compression, and transmission/storage, multiple buffers are needed. The small RAM (4KB to 10KB) and the requirement for multiple buffers limits the size of the individual trace buffers to a few hundred bytes. Second, reliable delivery of large buffers over an unreliable wireless multi-hop network in WSNs is expensive [34].

Existing WSN data compression algorithms such as SLZW [34] or PINCO [2] also achieve relatively poor performance as they can not exploit the rich repetitions in traces as compression is performed independently on small buffers as explained earlier. (We quantify the poor performance of these compression algorithms in Section 2.) Poor performance of established compression algorithms explains why existing WSN tracing approaches either use simple, ad hoc techniques [41], or do not compress at all [19, 37, 21].

1.5 Hybrid Trace Compression

This paper proposes *Prius* (named after the Toyota Prius hybrid car), a novel hybrid (offline/online) approach to compress WSN traces generated by various tracing frameworks. Prius relies on the following key observations:

- WSN computations exhibit a high degree of repetition in short time.
- The repetitive patterns in WSN computations evolve only little over time.
- WSN nodes use Harvard architecture and thus have separate program memory (EEPROM) and data memory (SRAM). While the latter memory is extremely scarce the former memory has more generous constraints and is rarely a bottleneck.

The key idea of Prius is thus to capture the repetitive patterns of WSNs that occur in the traces using an *offline training* and include those patterns in the *program memory* using *specially adapted* data structures. The compression algorithm then uses these patterns to perform *online compression*. While rather intuitive, our approach is based on a careful balance, which our evaluation validates. While program memory can accommodate more patterns than data memory and thus potentially improve compression ratio, accessing such memory is typically 1.5$\times$ more costly than accessing data memory in both ATMEL's AVR [15] and TI's MSP430 [30] architectures. The use of specialized data structures can counter-balance this increase by simplifying lookups considerably, but this does not support addition of patterns at runtime; such missing patterns may reduce compression performance. As we show, the energy savings obtained through higher compression rates outweigh the additional CPU costs. Furthermore, missing patterns are rare since WSN executions are repetitive and do not evolve much. More substantial changes in execution patterns arising from reprogramming a WSN can be handled by uploading a new set of patterns for the latest version of the software.

Our approach has several advantages. (1) By identifying patterns offline, the online phase is saved from doing heavy-duty mining. (2) Compression ratio is significantly improved by retaining the state information (dictionary) across small input buffers, which implies energy savings in transmission and storage of traces. (3) By storing patterns in the program memory — besides allowing more patterns to be stored — the precious RAM can be conserved for other components. (Moreover, flash storage technology used in program memory has improved more significantly in past years in terms of density and price than the SRAM technology used in data memory.) (4) The use of specific data structures reduces lookup time and saves space in storing the patterns. (5) Last but not least, our approach allows a wide range of established compression algorithms to be applied in the WSN context.

1.6 Contributions

In addition to pinpointing the reasons for poor performance of established compression algorithms on traces, the contributions of this paper are the following:

- We propose a novel hybrid (offline/online) trace compression technique that enables established compression algorithms to be applied in the WSN context.
- We describe two realizations of it by "hybridizing" the popular FCM and SLZW compression algorithms respectively, and present their implementations.
- We show that our hybrid approach respects the resource constraints of WSNs (5% average program memory overhead) whilst reducing energy consumption on average by 46% and 49% compared to straightforward online adaptations of established algorithms and state-of-the-art trace compression algorithm respectively. We also show preliminary results with sensor data, illustrating the potential of our techniques beyond traces.

1.7 Roadmap

Section 2 details challenges for compression in WSNs and motivates our approach. Section 3 presents our hybrid trace compression technique, and its realization in two compression techniques; Section 3.5 discusses their implementations in TinyOS. Section 4 illustrates performance benefits of our approach through empirical evaluation. Section 5 discusses various issues and Section 6 contrasts with related work. Section 7 draws conclusions.

2 WSN Opportunities and Challenges

In this section, we first present a brief overview of established compression algorithms and analyze the opportunities and challenges for trace compression in WSNs, motivating the design of our solution presented in Section 3.

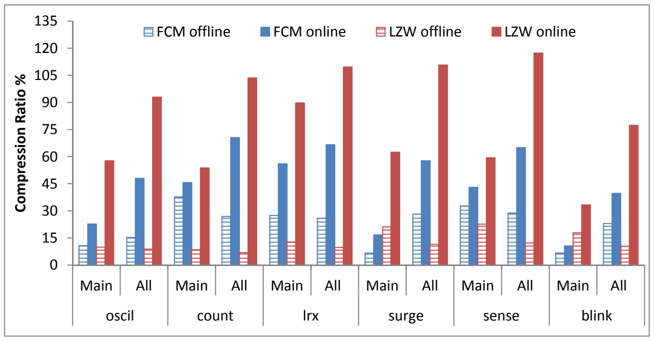


Figure 2. A comparison of online and offline compression ratios for FCM and LZW compression algorithms applied to control-flow traces generated by TinyOS applications. Smaller compression ratios are better.

2.1 Compression Algorithms

We present an overview of three of the most widely used (trace) compression approaches, namely *prediction*-based, *grammar*-based, and *dictionary*-based compression.

Prediction-based compression uses value predictors to compress a stream of values [7]. Only one or a few bits are needed to represent a value if the value can be correctly predicted. Otherwise, the original value is retained in the compressed stream. The prediction is based on the context table, which is updated as input is scanned. *Finite context methods* (FCM) [35] is a highly effective value prediction technique used for trace compression, which we describe in more detail shortly when applying our technique to it.

Dictionary-based compression algorithms build a dictionary of repetitive patterns by scanning the input values and compress the input by replacing the patterns with the indices to the dictionary. Named after its inventors Lempel, Ziv, and Welch, the LZW [46] algorithm is a well-known variant of the popular LZ family of text compression algorithms and is at the core of the Unix `compress` utility.

Grammar-based compression algorithms infer a grammar from the input text and produce that grammar as the compressed output [29, 24]. It has been shown that they are less effective than value prediction algorithms in trace compression [7]. Therefore, we focus on the two former families.

2.2 Offline vs. Online Compression

To understand how well the targeted traces can be compressed with standard compression techniques, we collected 30 minutes of control-flow traces for various TinyOS applications (see benchmarks in Section 4.3). We compressed these traces offline on a desktop using both FCM, which we implemented in Python, and Unix's `compress` implementation of LZW. We refer to these as FCM *offline* and LZW *offline* respectively. Note that offline compression uses the whole 30 minute trace as input and stores very large dictionaries/tables of patterns as it is running on a desktop.

Online compression, which is compressing traces as they are generated on the WSN nodes using straightforward adaptation of FCM and LZW, was done for comparison with offline compression. We implemented FCM in nesC respecting the resource constraints of WSNs and incorporated it into TinyOS applications, so the traces can be compressed online. For LZW, we used SLZW, an LZW implementation in nesC proposed by Sadler et al [34]. We collected the compressed traces for all the benchmarks. We refer to these as FCM *online* and LZW *online*. In contrast to offline compression, online compression uses small input buffers (192 bytes per buffer) and has limited dictionary/table storage as it runs on resource-constrained WSN nodes.

Figure 2 shows the compression ratio using FCM and LZW both offline and online compression for different TinyOS applications in our benchmark suite. From this figure we see that LZW offline can compress the trace down to 6.9%-22.72% of the original size, which represents a 4.4× to 14.5× reduction in size. Similarly, FCM offline can reduce the size from 6.69% to 37.77% of the original size, which represents 2.6× to 14.9× reduction in size. It is clear that the traces are well compressible, yielding an opportunity to save a considerable amount of energy.

The standard compression algorithms, however, do not work well for trace compression in WSNs if they are adapted *straightforwardly*. From Figure 2, we see that the compression ratio using LZW online is from 33.3% to 117% and for FCM online is from 10.5% to 70.5%. The compressed output can be larger than the input when the prediction is poor as encoding misprediction uses more bits than the original entry itself. We observe that there is a good scope for improvement for online compression (0.9× to 14× for LZW and 0.2× to 2.2× for FCM). For example, in the case of tracing all components in *oscil*, LZW offline can compress the trace 10.1×

more than LZW online and FCM offline can compress the trace 3.2× more than FCM online.

In summary, high compressibility of traces is an opportunity. However, applying established compression algorithms like LZW and FCM straightforwardly in WSNs results in a poor compression ratio. As explained in Section 1.4, the reason for poor performance is due to small input buffers and independence of compressed outputs, which allows independent decompression at the base station. Based on these observations, we present a novel generic hybrid trace compression approach in the following section.

3 Prius: Hybrid Trace Compression

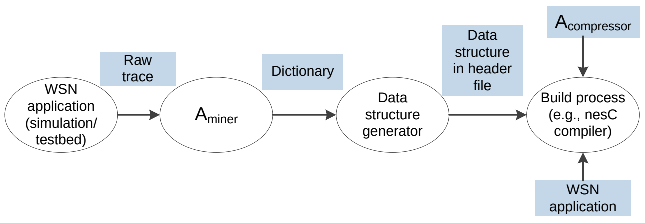
While the key idea underlying our approach is intuitive, its effective realization is less trivial. In this section, we first outline challenges in "hybridizing" a compression technique, before presenting the high-level design of our approach with respect to an abstract compression algorithm. Then we illustrate the intricacies of hybridization with respect to a specific compression algorithm by presenting the hybridization of FCM and LZW. Finally, we discuss our implementation details.

3.1 Hybridization Challenges

WSN computation is repetitive; it's repetitive nature can be effectively captured offline and the captured information can be used during the online compression of traces. Based on this observation, it is possible to design a hybrid compression that mines the patterns offline and stores them in the *data* memory. However, such an approach cannot improve compression ratio significantly because not many patterns can be stored in the limited data memory. Moreover, the space occupied by the patterns cannot be used by other components for the lifetime of the WSN application. Even when the *program* memory is used to store patterns, it is important to store them efficiently such that lookup is fast. Since the number of patterns in the program memory stored can be large, sequential scanning of all the patterns in the program memory to find a pattern is CPU-intensive and may undermine the energy savings obtained by better compression. Therefore, hybridization has to be carefully done.

3.2 High-level Design

The high-level design of our approach is explained with respect to an abstract compression algorithm, denoted as A. First, we develop two modified parts of the original compression algorithm A, namely A_{miner} and $A_{compressor}$. Given an uncompressed trace, the algorithm A_{miner} outputs the internal data structure that is used for compression to a file in addition to compressing the input like the original algorithm A. Depending on the compression algorithm, the internal data structure could either be a table or a dictionary.An efficient data structure is designed such that it exploits the *static nature of the patterns stored* to reduce access time and/or storage space. A data structure generator would take the output of algorithm A_{miner} and produce an encoding of the designed data structure in a header file. The algorithm $A_{compressor}$ is the version of algorithm A adapted to run on motes. Algorithm $A_{compressor}$ includes the header file as well as an interface with the designed data structure instead of the one used in algorithm A. The algorithm $A_{compressor}$ is lightweight because it doesn't need to identify, update or store patterns. Figure 3 presents our design as a workflow diagram.



Raw trace
Dictionary
Data structure in header file
A_compressor
WSN application (simulation/ testbed)
A_miner
Data structure generator
Build process (e.g., nesC compiler)
WSN application

Figure 3. Prius workflow.

3.3 FCM

Next we outline the FCM compression algorithm through an example and then present the steps to hybridize it, including the choice of an efficient data structure.

3.3.1 Description

FCM (Finite Context Methods) is a highly effective value prediction technique [7] that can be used for compression. A value is predicted based on a fixed number of preceding values, called the *context*. The number of preceding values, i.e., the size of the context, is configurable, and is added to the algorithm name. For example, if the context consists in 3 preceding values, the predictor is called *FCM-3*. A lookup table is maintained to store predictions corresponding to a limited number of context patterns encountered in the past.

Algorithm 1 Finite Context Machine (FCM-n) algorithm. Takes as $input$ a string of N characters and returns a compressed string as $output$. Assumes the presence of a table that stores the context and its corresponding prediction. Procedure PREDICT looks up the given context in the table. Procedure LEFTSHIFT left shifts the contents of the context once and appends the new input character. Procedure UPDATETABLE adds the context if it doesn't exist and otherwise corrects the prediction. Procedures APPENDCTXT, APPENDBYTE, APPENDBIT append the second argument to the first argument.

```
1:  for $i \leftarrow 0$ to $n$ do
2:      APPENDCTXT ($context$, $input[i]$)
3:      APPENDBIT ($output$, 0)
4:      APPENDBYTE ($output$, $input[i]$)
5:  end for
6:  for $i \leftarrow n+1$ to $N$ do
7:      if PREDICT ($context$)=$input[i]$ then
8:          APPENDBIT ($output$, 1)
9:      else
10:         APPENDBIT ($output$, 0)
11:         APPENDBYTE ($output$, $input[i]$)
12:         UPDATETABLE ($context$, $input[i]$) {Omit in hybrid}
13:     end if
14:     LEFTSHIFT ($context$, $input[i]$)
15: end for
```

Given a value i to compress, its context is used to find the prediction from the table. If i matches the prediction, a '1' bit is inserted to the compressed stream to indicate prediction success. If i does not match the prediction or the context does not exist in the lookup table, a '0' bit followed by i is added to the compressed stream and the lookup table is updated to reflect the new prediction. Decompression is straightforward when a lookup table is maintained. If the bit read is '0', the value is read from the input. Otherwise, the value is identi-

Table 1. FCM-3 example. Bits are represented with *overbar*. The input characters are 8 bits long.

Input	$ABCDECDECDECDE$
Output	$\bar{0}A\ \bar{0}B\ \bar{0}C\ \bar{0}D\ \bar{0}E\ \bar{0}C\ \bar{0}D\ \bar{0}E\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{1}$
FCM Table	$ABC \rightarrow D$ $BCD \rightarrow E$ $CDE \rightarrow C$ $DEC \rightarrow D$ $ECD \rightarrow E$

Table 2. LZW example. The input characters are 8 bits long. The output characters are 9-bits.

Input	$ABCDECDECDECEF$
Output	65 66 67 68 69 258 260 259 67 69 70
LZW Dictionary	$AB \rightarrow 256$ $BC \rightarrow 257$ $CD \rightarrow 258$ $DE \rightarrow 259$ $EC \rightarrow 260$ $CDE \rightarrow 261$ $ECD \rightarrow 262$ $DEC \rightarrow 263$ $CE \rightarrow 264$ $EF \rightarrow 265$

fied from the lookup table. The FCM compression algorithm is shown in Algorithm 1 and an example is shown in Table 1.

3.3.2 Hybridization

The hybridization of FCM involves designing an efficient data structure for the dictionary to be stored in the program memory and creating a hybrid version of the FCM algorithm shown above which accesses the table efficiently. The latter is simpler for FCM as the only change needed to Algorithm 1 is to omit line 12, which updates the dictionary. However, the procedure PREDICT has to be rewritten to access the table from the program memory.

The FCM table consists of entries with n-character contexts and their predictions. The table can be represented using a simple array or a hash table. An array is efficient for small tables as scanning the array may be quicker than calculating hash functions with complex mathematical operators. Furthermore, an array uses less space. However, a hash table is preferable for larger tables (100s of entries) as the lookup cost quickly adds up. Our evaluation considers both array-based and hash table-based implementations.

We observe that *the keys are static*. Therefore, we can build a hash table without collision. In other words, we can use *perfect hashing* [10], which is a double hashing technique that avoids collisions. However, implementing perfect hashing for WSNs is quite challenging.

3.3.3 Perfect Hashing for WSNs

We describe how we have adapted a well-known open-source implementation of perfect hashing library, GNU's `gperf`, for WSNs. For a given set of strings, GNU's `gperf` produces a hash function and hash table, in the form of C or C++ code. The main challenge is that the input character set for `gperf` can only be alphanumeric characters. The naïve approach of converting the integer ASCII value (e.g. 143) to a string (e.g. '143') was expensive due to CPU intensive division and `mod` operations. Another approach is to store every ASCII value of a byte in the form of a string, which is expensive in terms of space. We converted the integer into string in hexadecimal representation (e.g. '8f'), which uses only shift operations and a lookup of each nibble. This enabled perfect hashing with `gperf` for WSNs.

3.4 LZW

Next we outline the LZW compression algorithm through an example and then present the steps to hybridize it, including the choice of an efficient data structure.

3.4.1 Description

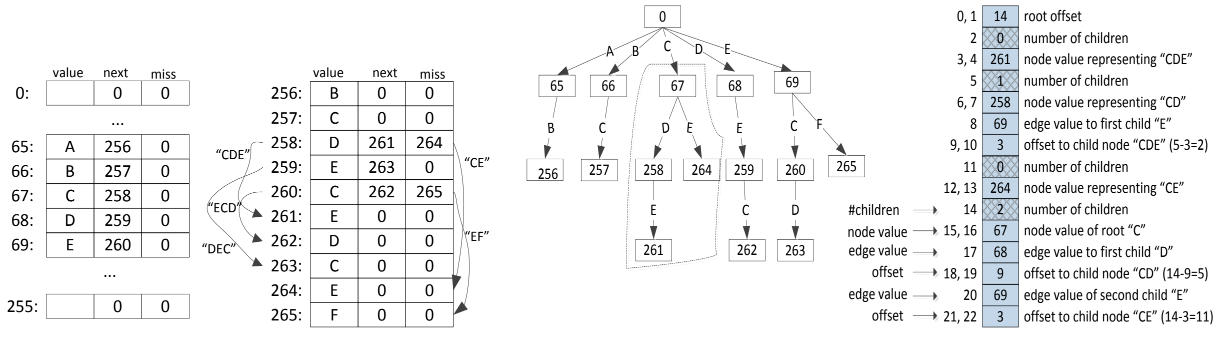
LZW [46] is a dictionary-based compression algorithm which builds a dictionary of repetitive patterns while scanning the input. The patterns found in the input are replaced (encoded) with indices to the dictionary. Since a pattern can be the prefix of other patterns, the pattern search continues until the longest pattern is found before encoding . New patterns are added to the dictionary. The LZW compression algorithm is shown in Algorithm 2 and an example is shown in Table 2. Decompression proceeds similar to the compression algorithm by maintaining a dictionary.

Algorithm 2 LZW algorithm. It takes a string of length N characters, *input*, and returns a compressed string in *output*. It assumes the presence of a dictionary that stores the pattern and its corresponding encoding. The dictionary is initially empty. Procedure LOOKUPDICTIONARY looks up the given context in the dictionary. Procedure ADDTODICTIONARY adds the pattern with a new encoding for that pattern. Procedure ENCODE returns the encoding of that pattern from the dictionary. Procedure APPEND appends the second argument to the first argument.

```
1:  $pattern \leftarrow input[0]$
2:  for $i \leftarrow 1$ to $N$ do
3:      $newPattern \leftarrow$ APPEND ($pattern$, $input[i]$)
4:      if LOOKUPDICTIONARY ($newPattern$) $\neq nil$ then
5:          $pattern \leftarrow newPattern$
6:      else
7:          APPEND ($output$, ENCODE ($pattern$))
8:          ADDTODICTIONARY ($dictionary$,$newPattern$) {Omit in hybrid}
9:          $pattern \leftarrow input[i]$
10:     end if
11: end for
12: APPEND ($output$, ENCODE ($pat$))
```

Implementing the LZW algorithm in WSNs is not straightforward – especially maintaining a dictionary and looking up arbitrarily long patterns. SLZW [34] is an efficient implementation of the LZW algorithm with an array-based data structure. Each entry in the array is a tuple (`value`, `next`, `miss`), in which, `value` stores the input character, `next` stores the pointer to the next entry in a pattern and `miss` refers to a new entry to further look for a matching pattern when the current pattern does not match. The dictionary initially contains 256 entries with each entry's `value` corresponding to its index and the `next` and `miss` pointers are initialized to 0.

Figure 4(a) shows an example of the array-based data structure for dictionary in Table 2. To illustrate the data structure, consider the patterns "AB" to "CE" found in the LZW dictionary in Table 2. To store "AB", an entry 256 is created with `value` "B" and a link is created from entry 65 to entry 256 by storing 256 in the next pointer of entry 65. While other patterns until "CE" are stored in a similar way, storing "CE" requires `miss` pointer. To store "CE", an entry 264 is created with value "E". Since the `next` pointer in en-


value next miss
0: 0 0
...
65: A 256 0
66: B 257 0
67: C 258 0
68: D 259 0
69: E 260 0
...
255: 0 0
256: B 0 0
257: C 0 0
258: D 261 264
259: E 263 0
260: C 262 265
261: E 0 0
262: D 0 0
263: C 0 0
264: E 0 0
265: F 0 0
"CDE"
"ECD"
"DEC"
"CE"
"EF"
0
A B C D E
65 66 67 68 69
256 257 258 264 259 260 265
261 262 263
0, 1 14 root offset
2 0 number of children
3, 4 261 node value representing "CDE"
5 1 number of children
6, 7 258 node value representing "CD"
8 69 edge value to first child "E"
9, 10 3 offset to child node "CDE" (5-3=2)
11 0 number of children
12, 13 264 node value representing "CE"
#children 14 2 number of children
node value 15, 16 67 node value of root "C"
edge value 17 68 edge value to first child "D"
offset 18, 19 9 offset to child node "CD" (14-9=5)
edge value 20 69 edge value of second child "E"
offset 21, 22 3 offset to child node "CE" (14-3=11)

(a) Array data structure (b) Trie data structure with compact encoding

Figure 4. Comparison of data structures used in LZW online and hybrid algorithms. Storage of some patterns is shown with arrows for clarity in Figure (a). Memory layout of the subtree rooted at node 67 is shown with shaded boxes along with the memory addresses on the left and encoding explanation on the right in Figure (b).

try 67, entry 258, is used to store "CD", the entry 264 has to be stored in the `miss` pointer of entry 258, thus creating a link between entry 67 and entry 264. To look up "CE", three lookups are needed. First, entry 67 is looked up. Since the value "C" matches, the next pointer, which is entry 258, is followed. Since the `value` of entry 258 is "D", the `miss` pointer, which is entry 264, is followed. Since the `value` of entry 264 is "E", the lookup correctly returns 264.

The key advantage of this data structure is that it allows to store partial matches succinctly as well as quickly determine if a longer patterns exists in the dictionary. LZW always looks for longer pattern by appending to the existing pattern in the dictionary. Suppose the pattern "CD" is matched, checking whether the pattern "CDE" is present will start looking for the presence of "E" directly from the entry 258, corresponding to "CD" instead of checking from entry 67 corresponding to "C", the beginning of the pattern.

3.4.2 *Hybridization*

The hybridization of LZW involves designing an efficient data structure for the dictionary to be stored in the program memory and creating a hybrid version of LZW algorithm shown above which accesses the table efficiently. The latter is simpler for LZW as the only change needed to Algorithm 2 is to omit line 8, which adds to the dictionary. However, the procedure LOOKUPDICTIONARY has to be rewritten to access the table from the program memory.

While the data structure described above for SLZW is quite efficient for online compression, it has a number of drawbacks when used for hybrid compression. First, since the patterns are known, the `next` and `miss` pointers storing 0 are unnecessary as no more patterns would to be stored. Second, when several patterns have common prefixes, the lookup cost of a pattern grows with the number of successors, which are patterns that have same common prefix but different current entries. For example, the patterns "CA", "CB", "CC" are successors of pattern "C". The lookup function has to iterate over the successors one at a time. Even if these are stored in some (ascending) order, binary search cannot be performed as they are stored as a linked list. We design an efficient data structure overcoming these issues.

3.4.3 *Compact Tries*

A prefix tree, or *trie*, is an ordered tree data structure that is used to store an associative array. A trie data structure for the dictionary used in the example is shown in Figure 4(b). The edges in the trie represent the input characters and the nodes represent the encoded dictionary values.

We observe that *the LZW dictionary is static* and exploit it for better performance as follows. First, since successors are known beforehand, only pointers to those successors are stored at any given node, thus avoiding `miss` pointers or `next` pointer with null values used in the array-based data structure described earlier. Second, we store the successor edges in *ascending* order to enable faster lookups using *binary search*. Finally, the trie can be compactly encoded (or tightly packed) in the memory. Such *compact tries* allow faster lookup of successors than array-based data structures by doing binary search on the children at a given node. Binary search is possible because the children can be stored at fixed offsets from each other allowing random access.

There are several ways to tightly pack a read-only trie in memory and we use one such efficient encoding presented by Germann et al. [12]. In this encoding the trie is represented bottom-up. Each node stores the number of children (1 byte), the node value (2 bytes), and then for each child, the edge value (1 byte) and the offset to that child. A complete memory layout of the encoding of subtree rooted at node 67 ("C") is shown in Figure 4(b). The beginning of each node is shown with a pattern filled box for clarity.

3.5 Implementation

We implemented the offline compression algorithms presented above in Python and C. For the online and hybrid versions of the algorithms, we used nesC version 1.3.2 and TinyOS 1.x. While implemented for TinyOS 1.x, our approach is OS agnostic and can be easily adapted to other WSN OSs including Contiki or SOS.

3.5.1 *FCM*

We implemented the offline version of FCM, in python and the online version of FCM, in nesC. We implemented hybrid versions of FCM, FCM_{miner} in Python and $FCM_{compressor}$ in nesC. We also implemented two variations of $FCM_{compressor}$, namely, Hybrid simple and Prius that use simple and efficient data structures to represent the table in program memory respectively. FCM_{miner} dumps the dictionary in a header file which is then converted into efficient data structure by a script such that Prius could use it. Hybrid simple simply uses the dumped header file as is.

3.5.2 LZW

We downloaded the SLZW code from [33]. SLZW has a mini-cache to reduce compression size further. We left the mini-cache on and allowed the input to expand. We used the SLZW code as is for the online version.

We implemented hybrid versions of SLZW similar to FCM. We modified SLZW code to create $SLZW_{miner}$, which does not use mini-cache, and dumps the dictionary to a file. Similarly, we created $SLZW_{compressor}$, that uses the dictionary from the file instead of creating its own dictionary. We implemented two variations of $SLZW_{compressor}$, analogous to $FCM_{compressor}$, namely, Hybrid simple and Prius.

4 Evaluation

Our evaluation demonstrates how our generic hybrid trace compression technique Prius enables the use of various well-known compression techniques in the WSN context. In particular we substantiate our previous claims, namely, (1) hybridization using program memory is effective and (2) efficient data structures are useful and in some cases mandatory for improving the effectiveness of hybridization.

4.1 Overview

We evaluated the previously outlined hybridized versions of the established dictionary-based compression algorithms FCM and SLZW in *nesC* for *TinyOS*. For each of these algorithms, we evaluated two variations of hybrid versions, namely, Hybrid simple and Prius that respectively use simple and efficient data structures to represent the dictionary/table in program memory respectively. For comparison, we implemented online (Online) and offline (Offline) versions of these algorithms if the implementations are not publicly available.

Prius can be applied to different types of runtime traces. We used the diagnostic concurrent interprocedural control-flow trace produced by the state-of-the-art tracing solution, TinyTracer [41, 42] that is publicly available and can record traces generated by multiple system components. TinyTracer includes a simple trace compression algorithm (TinyTracer), which is based on two simple techniques: (1) mining the top two frequent patterns of size up to 26 bytes offline and using those for online compression; (2) using run-length encoding. We compared Prius to TinyTracer quantitatively in this section and qualitatively in Section 6. For validity, we evaluated our compression techniques for other traces including another state-of-the-art tracing solution, LIS [37] and a real sensor dataset from environmental monitoring deployment [16]. Since Online algorithms gain with larger input buffers, we also evaluated large buffer effect.

We use four metrics – smaller values are always better:

(1) *Compression ratio* – quantifies the reduction in the trace size. It is defined as the ratio between compressed and uncompressed sizes and is represented as a percentage.

(2) *Energy overhead* – quantifies the increase in the amount of energy required to trace an application. It is defined as the additional energy required to trace an application and is represented as a percentage of energy consumed by the base application without tracing.

(3) *Program memory overhead* – quantifies the additional program memory required to hold the table of patterns mined offline. It is represented as a percentage of program memory required by the application with compression turned off.

(4) *Data memory overhead* – quantifies the additional RAM used.

Our main results show that Prius achieves high compression rate (up to 68% for FCM and 86% for LZW) and significant energy savings (up to 68% for FCM and up to 90% for LZW) compared to straightforward adaptations of compression algorithms. Similarly, Prius achieves high compression rate (up to 72% for FCM and 77% for LZW) and significant energy savings (up to 96% for FCM and 70% for LZW) compared to TinyTracer, the state-of-the-art WSN trace compression technique. The energy savings from writing less bytes to flash thus outweighs the overhead of running the compression algorithm or accessing program memory.

The program memory overhead due to storing the dictionary/table is modest (up to 24% for FCM, 20% for LZW). The data memory overhead is due to memory buffers used to store the inputs, compressed outputs, and the dictionary to store patterns. The buffers for storing inputs and compressed outputs are the same for both hybrid and online (Online) compression techniques. However, the dictionary or table in the online algorithms (2KB for SLZW and 0.5KB for FCM) use precious RAM, whereas, Prius (all hybrid) doesn't incur this overhead and thus we don't discuss this metric further.

4.2 Evaluation Methodology

We used TOSSIM for the reported results due to the difficulty of performing energy measurements directly on the hardware and problems with emulators. Avrora has well-known problems in flash energy estimation. ATEMU [31] emulations showed to be problematic when accessing program memory for large programs and no support is available.

We implemented hybrid versions of both the FCM and SLZW algorithms in nesC for TinyOS 1.x and integrated it with TinyTracer so our compression implementation is used to compress traces instead of the default compression in TinyTracer. We collected the uncompressed trace for 15 minutes by simulating each of the benchmarks in TOSSIM for a simple 4-node network. The trace is stored in the flash at the end of the simulation. This raw trace was used to get Offline results and for mining the patterns for hybrid versions. For mining patterns, we used the first half of these traces. To get the compressed results for Online, TinyTracer, Hybrid simple and Prius, we simulated the application for 15 minutes using TOSSIM for a 4-node network in grid topology. Since the benchmark application repeats every few seconds, 15 minutes are representative of the long-time behavior for the application.

We used PowerTossim [38] to measure the energy consumption of the simulation. Since TOSSIM doesn't distinguish between program memory and data memory, the energy overhead in accessing program memory is measured as follows. Every program memory access requires 1.5 CPU cycle whereas access to data memory takes 1 CPU cycle in ATMEL AVR architectures. Therefore, we instrumented the code to count the number of memory accesses to the table in program memory. We incremented the CPU cycle count by 0.5 times the number of memory accesses in PowerTossim results. The traces are obtained from the flash file. We used a trace parser that measures the size of the traces.

4.3 Benchmark Suite

For our evaluation, we chose as benchmarks five default TinyOS 1.x applications that are widely studied by others [9, 37, 41], as well as a large TinyOS application, `LRX`, which is a module for reliable transfer of large data developed as part of the Golden Gate Bridge monitoring project and is one of the largest nesC components ($\sim$ 1300 lines of nesC code) in TinyOS 1.x. We used `SingleHopTest` to drive the `LRX` module. These six benchmarks are described in Table 3.

TinyTracer [41] allows tracing multiple nesC components at the same time and can handle high throughput trace generation. To study the effect of compression on larger traces, we traced all the nesC components included in an application and these include the main component (e.g., `SurgeM` for Surge) as well as the system components such as LEDs (e.g., `LedsC`), sensor (e.g., `PhotoTempM`), radio (e.g., `AMStandard` or `MultihopEngineM`), and timer (e.g., `TimerM`). For every benchmark, we traced all the components starting with one component and gradually adding additional components until all components are traced. Table 3 shows the system components used by each benchmark. The order in which the components were included in the simulation is the following: main, LEDs, sensor, radio or network layer, and lastly timer. For example, '`surge-1c`' in the results means just the main component is being traced whereas '`surge-3c`' means that the main, LED, and sensor components are being traced.

4.4 FCM

Figures 5, 6, and 7 respectively show the compression ratio, energy overhead and program memory overhead for various online (Online), hybrid (TinyTracer, Hybrid simple, Prius) and offline (Offline) versions of FCM compression algorithm applied to control-flow traces generated by TinyTracer for TinyOS applications in our benchmark suite.

4.4.1 *Effect of Program Memory*

We first observe from Figure 5 that both Hybrid simple and Prius compress the input very well compared to Online, showing that hybridization pays off. The improvement in compression ratio for Prius over Online ranges from 18% (`lrx-2c`) to 68% (`count-4c`) and the average improvement over all benchmarks is 45%. In other words, the traces produced by Online are 22% to 215% (average 102%) bigger than the traces produced by Prius for the benchmarks. Hybrid simple also shows similar improvement of compression ratio over Online. This is due to the many patterns that can be stored in the dictionary or table in the program memory as opposed to data memory used by Online algorithms.

Similarly, the improvement in the compression ratio for Prius over TinyTracer is on average 31% and up to 72% (`surge-5c`). In other words, the size of the trace generated by TinyTracer is on average 66% larger and up to 252% (`surge-5c`) larger. When tracing only one or fewer component(s), for benchmarks like `blink` or `surge`, TinyTracer generates up to 17% (`surge-1c`) smaller traces than Prius because the trace consists mostly of top two frequent patterns used by TinyTracer. However, as the number components traced increase, many patterns appear in the trace making TinyTracer's compression largely ineffective. Thus, this result substantiates the claim that many patterns need to be stored to get effective compression and since data memory is precious, program memory needs to be used.

The improvement in compression ratio translates to considerable energy savings for Prius as shown in Figure 6. The reduction in energy overhead due to Prius over Online ranges from 17% (`sense-1c`) to 68% (`oscil-5c`) and the average is 46%. Similarly, the reduction in energy overhead due to Prius over TinyTracer is up to 96% (`surge-1c`) and the average is 49%. The average reduction in energy overhead due to Prius over Online and TinyTracer increases to 56% and 59% respectively when all components are traced. The reason for energy savings is that smaller traces result in fewer writes to flash storage and since energy required to write to flash is orders of magnitude more than CPU, saving flash writes conserve energy significantly.

The cost of hybrid approaches is in the program memory overhead, which is shown in Figure 7. The increase in program memory due to Prius over Online ranges from 0.8% (`surge-1c`) to 32% (`lrx-4c`) and the average increase over Online across all benchmarks is about 6%. Similarly, the increase in program memory due to Prius over TinyTracer ranges from 0.7% to 32% and the average is 5%. We observe that Hybrid simple does not incur much overhead with average only 0.2% and maximum of 7%. We think the reduction in compiler inlining has compensated the increase due to the dictionary. However, the cost of lost inlining is insignificant as can be seen from the energy overhead results.

We observe that the compression ratio obtained by hybrid techniques Prius and Hybrid simple is close to the Offline compression ratio as most of the patterns used by Offline are known to hybrid techniques as well. It is interesting to note that for some benchmarks (e.g., `surge`), hybrid versions have slightly better compression ratio than Offline. This is because the offline algorithm takes some time to learn the patterns from the input and therefore, first several entries in the input are not compressed. In contrast, Prius and Hybrid simple start compressing from the first input character.

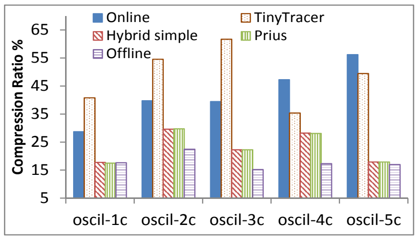
4.4.2 *Effect of Perfect Hashing*

While Hybrid simple compresses traces as well as Prius, Prius saves energy over Hybrid simple because perfect hashing reduces lookup time considerably. The energy overhead reduction of Prius over Hybrid simple is on average 30% and up to 78% (`lrx-4c`). Hybrid simple is competitive ($\sim$10%) when only one or few components are traced as the number of patterns in such cases is small. For one benchmark (`sense-1c`), Hybrid simple even reduces energy overhead over Prius by 12%. When the number of components traced and thus the number of patterns increases though, the efficient datastructure in Prius clearly reduces energy overhead.

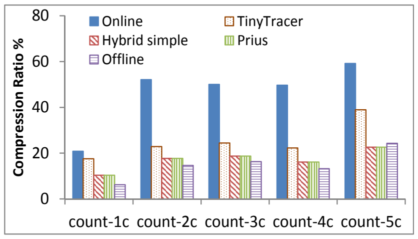
The average reduction in energy overhead due to Hybrid simple over Online is only 17% despite much higher com-

Table 3. The TinyOS 1.x applications in our benchmarks suite. C LOC is the lines of C code generated by nesC compiler.

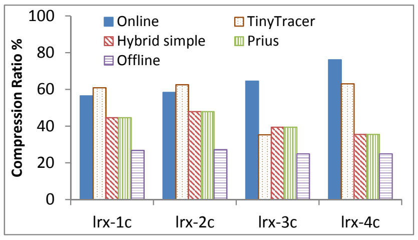
TinyOS application	Alias	Description	Period (s)	C LOC	System components used
`Blink`	`blink`	Toggle the LEDS	1	2061	LEDs, timer
`Sense`	`sense`	Samples sensors and displays it on LEDS	0.5	3730	LEDs, sensor, timer
`Oscilloscope`	`oscil`	Data collection with high sensing rate	0.125	5956	LEDs, sensor, UART, timer
`Surge`	`surge`	Data collection with medium sensing rate	2	11358	LEDs, sensor, radio, timer
`CntToLedsAndRfm`	`count`	A counter that broadcasts and displays count	0.25	8241	LEDs, sensor, radio, timer
`LRX`	`lrx`	Reliable large data transfer application	2	10015	LEDs, radio, timer



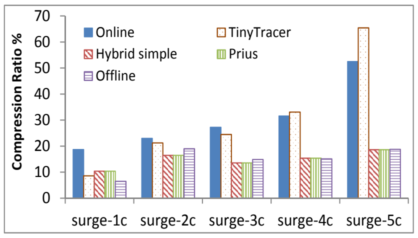
(a) `oscil`



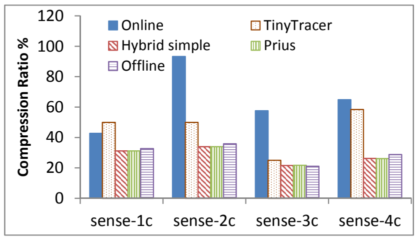
(b) `count`



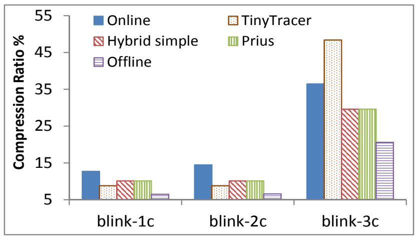
(c) `lrx`



(d) `surge`

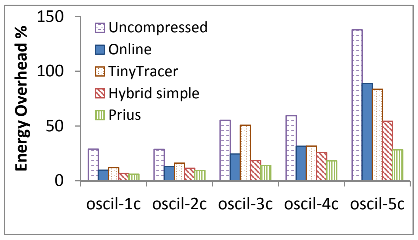


(e) `sense`

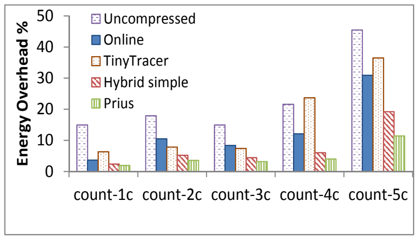


(f) `blink`

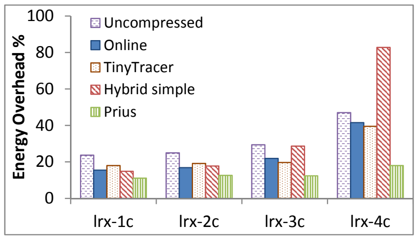
Figure 5. Compression ratio for various online, hybrid and offline versions of FCM compression algorithms applied to control-flow traces generated by TinyOS applications. The smaller the compression ratio, the better the compression is.



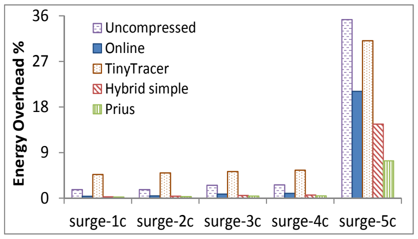
(a) `oscil`



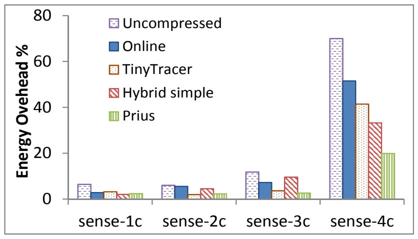
(b) `count`



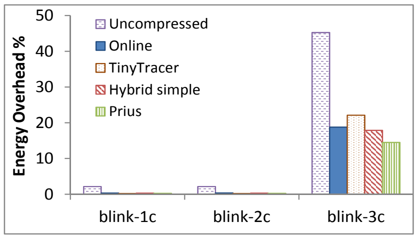
(c) `lrx`



(d) `surge`



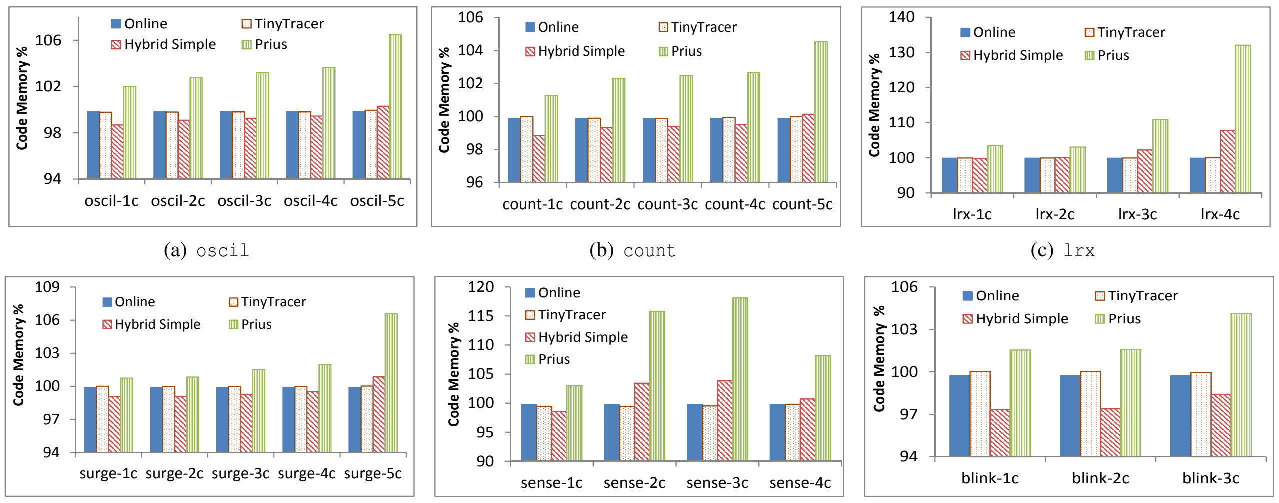
(e) `sense`



(f) `blink`

Figure 6. Energy overhead for various online and hybrid versions of FCM compression algorithms applied to control-flow traces generated by TinyOS applications.

pression achieved by Hybrid simple. The reason is that sequential scanning of a program memory table can be very CPU-intensive for large tables. For example, benchmark `lrx-4c` is a *degenerative case* for Hybrid simple because the energy overhead was larger than the energy overhead of uncompressed tracing. `lrx-4c` is the most complex benchmark in our suite and has a large number of patterns. However, even when `lrx-4c` is regarded as an outlier and omitted, the average reduction in energy overhead of Prius over Hybrid simple is 27%. Thus, we see that efficient data structures are very helpful for effective hybridization and in fact, mandatory for cases like `lrx-4c`.



(a) `oscil` (b) `count` (c) `lrx`

(d) `surge` (e) `sense` (f) `blink`

Figure 7. Program memory overhead for various online and hybrid versions of FCM compression algorithms applied to control-flow trace generated by TinyOS applications.

4.5 LZW

Since the results for the LZW compression algorithm follow the same trend as FCM and in the interest of space, we show the results for all benchmarks in the case when all the application components are traced. Figure 8 shows the compression ratio, energy overhead and program memory overhead for various online (Online), hybrid (TinyTracer, Hybrid simple, Prius) and offline (Offline) versions of the SLZW compression algorithm applied to control-flow trace generated by TinyOS applications in our benchmark suite. In LZW, the input could expand if there are not enough repetitions and this happened for a few benchmarks for Online.

4.5.1 *Effect of Program Memory*

The improvement in compression ratio for Prius over Online ranges from 74% (`lrx-4c`) to 86% (`count-5c`) and the average improvement over all benchmarks when all components are traced is 81%. The reason for this improvement is the limited dictionary size in the Online algorithm. This improvement in compression ratio translates to considerable energy savings ranging from 82% (`lrx-4c`) to 90% (`count-5c`) and the average energy savings is about 85% over all the benchmarks. Hybrid simple showed similar compression ratio (average of 81%) and energy savings (average of 80%). The improvement in compression ratio for Prius over TinyTracer ranges from 72% (`lrx-4c`) to 77% (`surge-5c`) and the average is 75% over all benchmarks. This improvement in compression ratio translates to considerable energy savings ranging from 55% (`lrx-4c`) to 70% (`surge-5c`) and the average is 64%.

The program memory overhead for Prius over Online for all the benchmarks ranges from 0.3% (`blink-3c`) to 26% (`oscil-5c`) and the average is 13%. For Hybrid simple, it ranges from 10% (`surge-5c`) to 29% (`oscil-5c`) and the average is 19%. Similarly, the average program memory overhead for Prius over TinyTracer is 11%. Thus, the program memory increase is modest.

4.5.2 *Effect of Compact Tries*

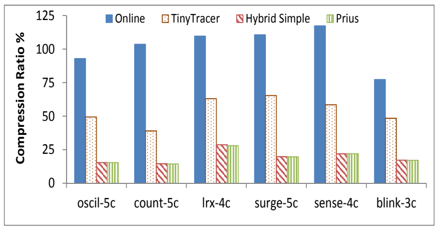
Prius saves energy and program memory overhead when compared to Hybrid simple as expected. The energy savings for Prius over Hybrid simple over all benchmarks when all components are traced ranges from 4% to 33% and the average is 24%. The average program memory savings for Prius over Hybrid simple ranges from 2% to 10% and the average is 5%. Savings are limited due to the fact that Hybrid simple already uses an efficient data structure unlike FCM, which uses a naïve datastructure based on a simple array. If a naïve array is used in SLZW, significant savings can be obtained as it is expensive to find longest matches in an array of patterns.

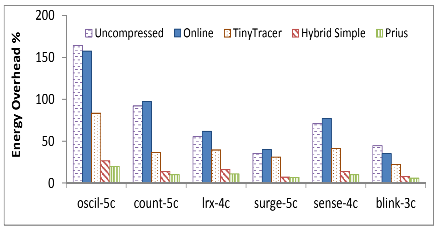
4.6 Effect of Larger Input Buffers

We have set the input buffer size to be 192 bytes in the results discussed so far. We recall that three input buffers are needed for tracing and compression. Two buffers are required to store the trace as it is generated. When one buffer is full, the other buffer stores the trace generated and the filled buffer is compressed. A third buffer is used to store the compressed output before it can be stored in the flash/sent on the radio. The RAM size and the need for three buffers forces each individual buffer size to be only few hundred bytes. In this section, we show that even if the size of input buffer is doubled, the conclusions still hold.

We study the effect of input buffer sizes of 288 and 384 bytes. As noted above, the total increase in RAM requirements will be thrice the input buffer size. When the input buffer size is increased, the online compression algorithm will perform better as larger inputs provide more chance to identify and replace patterns in the buffer. However, it will not affect hybrid algorithms that much. Therefore, we compared Online with two larger input buffers (Online 288 and Online 384) against Prius with 192 bytes. We used FCM and the results are shown in Figure 11 and Figure 10.

We observe that the increase in buffer size from 288 (Online 288) to 384 (Online 384) bytes improves the com-


(a) Compression tatio


(b) Energy overhead

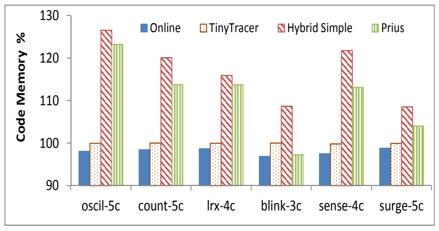

(c) Code memory

Figure 8. Compression ratio, energy overhead and program memory overhead of LZW compression algorithms applied to control-flow trace generated by TinyOS applications.

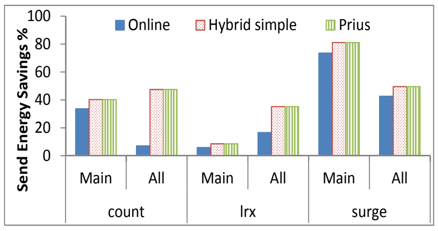


Figure 9. Energy savings for transmitting FCM compressed traces.

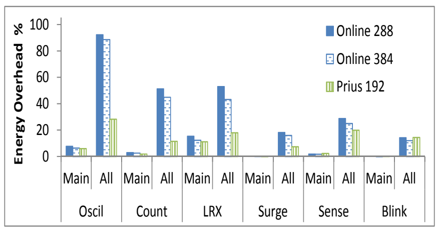


Figure 10. Energy overhead for traces compressed in large buffer.

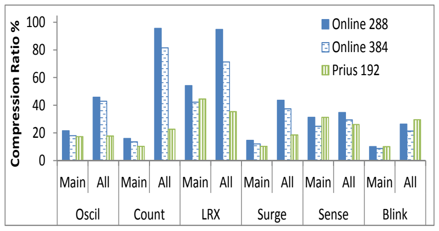


Figure 11. Compression ratio for traces compressed in large buffer.

pression ratio up to 25% and energy overhead up to 20%. Except for `blink-All`, the improvement in compression ratio due to Prius over Online 288 ranges from 18% to 76% and the reduction in energy overhead ranges from 23% to 78%. Similarly, except for a few benchmarks (`blink`, `sense-Main`, `lrx-Main`), the improvement in compression ratio due to Prius over Online 384 ranges from 3.16% to 72.21% and the reduction in energy consumption ranges from 9% to 75%.

Online performed better on benchmarks with few patterns. For those benchmarks, the improvement in compression ratio due to Online 384 over Prius ranges from 5% (`lrx-Main`) to 28% (`blink-All`) and reduction in energy overhead ranges from 12% (`blink-Main`) to 24% (`sense-Main`).

From these results, we conclude that while increased input buffer sizes help Online, Prius performs much better than Online 384 in many cases even with 192 bytes input buffer.

4.7 Radio Transmission Energy

Next we study the overhead of sending traces over the radio as opposed to writing to flash. We show that hybrid compression algorithms save more transmission energy than online compression algorithms.

We estimate the crucial energy overhead, which is the energy used by the radio for transmitting traces from a node instead of the total radio energy. By excluding the energy overhead required to obtain a radio channel, which varies depending on the network traffic and environment or the energy spent by the packet in the network stack, we can have a fair comparison between the two compression algorithms.

We used the power logs generated by PowerTOSSIM to estimate the transmission energy overhead. More precisely, we found the time intervals when the radio is in transmitting state and used the energy model for mica2 motes available in PowerTOSSIM for determining the current consumption.

Figure 9 shows the savings obtained by compression in the transmission energy overhead due to tracing. Each bar represents the percentage of energy savings over uncompressed tracing energy. In the interest of space, we only show for two configurations of benchmarks that use radio.

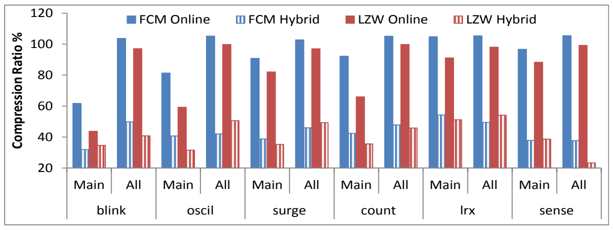
We first observe that the savings from any compression ranges from 7% to 81%. The average savings due to Online is 29% and due to Prius or Hybrid simple is 43.5%. Next, we note that the savings due to hybrid compression Prius and Hybrid simple over online compression Online is 2.77% (`lrx-1c`) to 43.5% (`count-5c`) and the average is 19.8%. The Prius and Hybrid simple have same energy savings because they compress the trace equally well and the CPU energy overhead is not included in this metric. We note that the energy savings shown are for a single node and the savings add up at every hop if the trace is transmitted over a multi-hop path to the base station.

4.8 Genericity

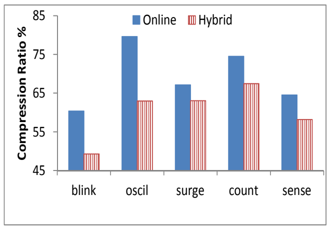
To assert the benefits of our techniques beyond control-flow tracing, we applied them on different kinds of traces and sensor data. Since we are interested in compression ratio, we simulated online and hybrid algorithms on PC as follows. We implemented both the Online and Hybrid compression algorithms as well as a small input simulator (SIS) in Python. SIS splits the input trace into multiple small files of 192 bytes (last file may be less than 192) and feed these to compression algorithms and combines the compressed output to form the final compressed file. It uncompresses the final compressed file output to verify the result. We used the first half of the trace to mine patterns offline for hybrid algorithms.

4.8.1 LIS

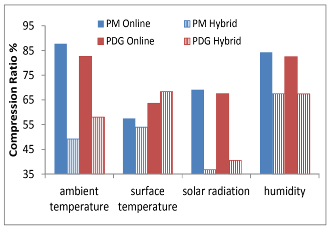
Log Instrumentation Specification (LIS) [37] is a runtime logging framework designed specifically for WSN debugging. It provides a language and runtime to gather runtime information efficiently by using local namespaces and bit-aligned logging. LIS has built-in support to capture the function call as well as intraprocedural control-flow paths.



(a) LIS traces



(b) Event traces



(c) Sensor data set

Figure 12. Compression ratio for LIS and event traces as well as sensor data from glacier monitoring deployment.

We ported LIS to work on TinyOS 1.x and modified it to write the trace into flash. We used ATEMU emulator to collect the uncompressed function call and intraprocedural control-flow traces. We used SIS to compress the traces using both FCM and LZW algorithms. Due to space, we show the results only for one component and all components cases.

The compression ratio results are shown for hybrid (Hybrid) and online (Online) algorithms as shown in Figure 12(a). We observe that Hybrid approach improves the compression ratio significantly. For FCM, the improvement in compression ratio due to Hybrid over Online ranges from 48.17% (`lrx-Main`) to 64.15% (`sense-All`) and the average is 54.74%. Similarly, for LZW, the improvement ranges from 20.64% (`blink-Main`) to 76.48% (`sense-All`) and the average is 50.14%. Based on our earlier results, we conclude that the low compression ratio reduces the energy overhead due to tracing significantly.

4.8.2 Event Traces

Event traces have been used for WSN debugging [19]. We manually instrumented code to record events such as the ones used in Dustminer [19]. An event contains a timestamp, event id, and parameters. Unlike control-flow traces event traces have data values and the timestamps associated with the events, which reduce the opportunities for compression.

We recorded asynchronous events, timer events, message events and any important events inside tasks by manually instrumenting the code for 15 minute runs of all benchmarks except lrx, which is too big to manually instrument. We compressed the trace using SIS and LZW algorithm to get the compression ratio, which is shown in Figure 12(b). We observe that the improvement in compression ratio due to Hybrid over Online ranges from 6% to 21% and the average is 13%. While the improvement is modest, such improvement is obtained for highly varying data.

4.8.3 Sensor Data Compression

In addition to execution traces, Prius can be applied to other kinds of data such as sensor data. Our initial resuts are encouraging. Often times, the sensor data can be compressed well using lossy domain specific techniques such as averaging or discarding values within thresholds. We observe that Prius complements these techniques and can be applied if the sensor data does not evolve significantly

We used two sensor data sets collected from two glacier monitoring deployments, namely, Plaine Morte glacier (PM) and Patouilee des glacier (PDG) using SensorScope in 2007/08 by Ingelrest et al. [16]. Both data sets contained data from 4 sensors, namely, *ambient temperature*, *surface temperature*, *solar radiation*, *humidity*. PM deployment had 13 locations while PDG deployment had 9 locations.

We compressed all four sensor data using SIS and computed the average compression ratio across locations for each sensor. We used LZW algorithm as FCM is not very effective for data compression. The results in Figure 12(c) show that except for PDG *surface temperature*, Hybrid compresses the sensor data very well. The improvement in compression ratio due to Hybrid over Online for *solar radiation* is 40% (PDG) and 47% (PM), for *ambient temperature* is 30% (PDG) and 44% (PM), for *humidity* is 18% (PDG) and 20% (PM), and for *surface temperature* it is 6% (PM) and -7% (PDG). For PDG *surface temperature*, Hybrid reduces the compression ratio, which could be due to anomalous data.

5 Discussion

Trace Divergence: If the trace generated is not present in the program memory table, then no compression happens and it is encoded as misprediction, which usually costs more bits than the entry itself. As our results show, mispredictions are rare. However, when the trace significantly diverges from the table in program memory and compression ratio falls below a threshold, a small footprint online compression techniques such as SLZW can act as a second layer compression.

Spatial Correlation: Our approach currently does not exploit the significant spatial correlation present in the network as most nodes do similar tasks in a WSN. Pioneering techniques [32, 11] exploit such correlations to further improve compression ratio.

Network Effect: When the traces are transmitted on the radio, the local energy savings due to compression multiplies with every hop between the node and the base station. Since the reliability of a multi-hop network decreases significantly with the number of hops, retransmissions are not uncommon. By sending compressed data, the number of packets transmitted including retransmissions can be reduced significantly resulting in huge energy gains and less network congestion, as also noted by Sadler et al. [34].

6 Related Work

Several trace compression techniques have been studied in software engineering literature. The main ones include the *value prediction algorithms* (VPC) [7, 8] such as FCM-based ones [35] [13]. Grammar-based text compression such as Sequitur [29] has been used successfully for compressing traces [24]. The widely used Unix utility programs `gzip`

(*LZ77*) and `compress` (*LZW*) have also been used for trace compression. However, these techniques are inapplicable for WSNs due to extreme resource constraints. Our hybrid approach enables the use of these techniques for WSNs.

Offline compression algorithms — algorithms that produce compressed output only after seeing the complete input — such as the ones proposed by Apostolico and Lonardi [1] and by Larsson and Moffat [23] explore how to reduce the compression ratio by compressing the whole file instead of using online techniques such as SLZW [34]. In contrast, our technique does not perform offline compression but uses an offline phase to mine the patterns and compress them during an online phase. Offline compression is not applicable to traces because traces are generated with program execution and are not known beforehand.

Specialized compression algorithms have been similarly proposed for different scenarios in the context of embedded systems. One group [4, 40] of work focused on energy as a metric for compression and profiled various off-the-shelf algorithms. In contrast, our work proposes a novel hybrid compression technique that exploits program memory size. Another body of research focused on adapting standard compression algorithms such as LZW to resource-constrained embedded devices [25, 27]. However, these techniques are targeted at devices which still have much more memory than sensor nodes and do not have an offline phase to exploit program memory. One piece of work that comes close to our approach is by Netto et al. [28]. The authors use profiling for code compression. The similarity between that approach and ours is the use of a static dictionary. Unlike our work, Netto et al. however target code compression and do not make use of program memory to store large numbers of patterns as Prius does. Moreover, our work is generic and can be applied to any dictionary-based approach. Code compression has been studied in WSNs [45] to reduce reprogramming cost. Such techniques aid Prius as they reduce the impact of large dictionaries in the program memory on reprogramming cost.

Several algorithms have been proposed in the context of WSNs for sensor data compression. Early pioneering work [32, 11] in this area exploited the high spatial correlation in dense networks. There have been several approaches that use in-network data aggregation [17, 2, 3]. These efforts are orthogonal to our work as our work exploits temporal correlation in traces; we could make further use of such approaches for spatial correlation.

Sadler et al. [34] proposed SLZW, a generic data compression algorithm — an adaptation of LZW to sensor nodes — and novel ideas to handle resource constraints such as mini-cache and data transforms. SLZW can handle varying data well. However, when applied to traces, it fails to capitalize on the rich amount of repetition; the major reason for this is the limited memory buffer. Furthermore, SLZW has high RAM requirements, e.g., in addition to input buffers 2KB of RAM is required to store the dictionary. Our approach is designed specifically to exploit many such repetitions.

We earlier proposed TinyTracer [41], an interprocedural control-flow tracing of concurrent events and a simple trace compression approach. The only commonality between our approach and TinyTracer is the idea of mining patterns offline and using them for online compression. However, there are three major differences between the two works, which leads to significant benefits of our approach as demonstrated in Section 4. (1) We use program memory to store patterns mined offline as opposed to TinyTracer [41], which stores only two patterns in the data memory. (2) Determining the set of patterns that yields minimal compression size when patterns overlap is shown to be NP-complete [39]. Unlike the ad hoc heuristic of using the top two patterns [41], our approach uses the table mined by established algorithms (e.g., LZW, FCM), which is more effective for trace compression as shown by our results. (3) Our approach uses efficient data structures suited for the compression technique (FCM, LZW) to conserve energy, while TinyTracer [41] does not use efficient data structures and performs naïve pattern matching.

7 Conclusions

We exploited the fact that WSN computations are highly repetitive and do not evolve much over time to propose a novel generic hybrid trace compression technique, called Prius that is suitable for WSNs. Our technique relies on using an offline training phase to learn repeating patterns and use it to drive the online compression. While intuitive, effective realization of it require using program memory to store the mined patterns as well as efficient data structures for faster inexpensive lookups. Our results show that Prius yield significant energy savings over straightforward adaptations of established compression techniques as well as state-of-the-art trace compression technique.

As a future work, we are investigating techniques that can slightly adapt the patterns mined offline to cope with any significant changes at run-time. More specifically, the idea is that the compression algorithm in the online phase dynamically triggers a pattern adaptation engine if the compression ratio drops below a given threshold.

8 Acknowledgement

This research is supported, in part, by the National Science Foundation (NSF) under grant 0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

9 References

[1] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. In *Proceedings of the IEEE*, 88 (11), 2000.

[2] T. Arici, B. Gedik, Y. Altunbasak, and L. Liu. Pinco: A pipelined in-network compression scheme for data collection in wireless sensor networks. In *12th IEEE International Conference on Computer Communications and Networks (ICCCN '03)*, 2003.

[3] S. J. Baek, G. D. Veciana, and X. Su. Minimizing energy consumption in large-scale sensor networks through distributed data compression and hierarchical aggregation. In *IEEE Journal on Selected Areas in Communications*, 22 (6), 2006.

[4] K. Barr and K. Asanović. Energy aware lossless data compression. In *ACM Trans. Comput. Syst*, 24 (3), 2006.

[5] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *6th ACM Conference on Embedded Networked Sensor Systems (SenSys '08)*, 2008.

[6] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle. Deployment techniques for wireless sensor networks. In *Sensor Networks: Where Theory Meets Practice, Springer*, 2009.

[7] M. Burtscher. VPC3: A fast and effective trace-compression algorithm. In *SIGMETRICS Perform. Eval. Rev.*, 32 (1), 2004.

[8] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, 2003.

[9] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *5th ACM International Conference on Embedded Networked Sensor Systems (SenSys '07)*, 2007.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms, 3rd Edition*. In *The MIT Press*, 2009.

[11] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution storage for sensor networks. In *1st ACM International Conference on Embedded Networked Sensor Systems (SenSys '03)*, 2003.

[12] U. Germann, E. Joanis, and S. Larkin. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP '09)*, 2009.

[13] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: increasing value prediction accuracy by improving table usage efficiency. In *7th International Symposium on High-Performance Computer Architecture (HPCA '01)*, 2001.

[14] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber. Wireless sensor networks in Permafrost research - concept, requirements, implementation and challenges. In *9th International Conference on Permafrost (NICOP '08)*, 2008.

[15] M. Horton, D. Culler, K. S. J. Pister, J. Hill, R. Szewczyk, and A. Woo. Mica: The commercialization of microsensor motes. In *http://www.sensormag.com/*, 2002.

[16] F. Ingelrest, G. Barrenetxea, G. Schaefer, and M. Vetterli, and O. Couach and M. Parlange SensorScope: Application-specific sensor network for environmental monitoring. In *ACM Trans. Sen. Netw.*, 06 (02), 2002.

[17] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. In *IEEE/ACM Trans. Netw.*, 11 (1), 2003.

[18] M. Keller, J. Beutel, A. Meier, R. Lim, and L. Thiele. Learning from sensor network data. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*, 2009.

[19] M. M. H. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *6th ACM Conference on Embedded Networked Sensor Systems (SenSys '08)*, 2008.

[20] M. M. H. Khan, H. K. Le, M. LeMay, P. Moinzadeh, L. Wang, Y. Yang, D. K. Noh, T. Abdelzaher, C. A. Gunter, J. Han, and X. Jin. Diagnostic powertracing for sensor node failure analysis. In *9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '10)*, 2010.

[21] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *5th International Conference on Mobile Systems, Applications and Services (MobiSys '07)*, 2007.

[22] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *20th International Conference on Parallel and Distributed Processing (IPDPS '06)*, 2006.

[23] N. Larsson and A. Moffat. Off-line dictionary-based compression. In *Proceedings of the IEEE*, 88 (11), 2000.

[24] J. R. Larus. Whole program paths. In *ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*, 1999.

[25] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *39th Annual Design Automation Conference (DAC '02)*, 2002.

[26] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *1st ACM International Conference on Embedded Networked Sensor Systems (SenSys '03)*, 2003.

[27] C. H. Lin, Y. Xie, and W. Wolf. LZW-based code compression for VLIW embedded systems. In *IEEE Conference on Design, Automation and Test in Europe (DATE '04)*, 2004.

[28] E. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Mixed static/dynamic profiling for dictionary based code compression. In *IEEE International Symposium on System-on-Chip*, 2003.

[29] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. In *The Computer Journal*, 40 (2 and 3), 1997.

[30] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, 2005.

[31] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. ATEMU: A fine-grained sensor network simulator. In *1st Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '04)*, 2004.

[32] S. Pradhan, J. Kusuma, and K. Ramchandran. Distributed compression in a dense microsensor network. In *IEEE Signal Processing Magazine*, 19 (2), 2002.

[33] C. Sadler. SLZW implementation. `https://sites.google.com/site/cmsadler/`, 2007.

[34] C. M. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *4th ACM International Conference on Embedded Networked Sensor Systems (SenSys '06)*, 2006.

[35] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '97)*, 1997.

[36] R. Shea, M. Srivastava, and Y. Cho. Optimizing bandwidth of call traces for wireless embedded systems. In *IEEE Embedded Systems Letters*, 1 (1), 2009.

[37] R. Shea, M. Srivastava, and Y. Cho. Scoped identifiers for efficient bit aligned logging. In *Design, Automation Test in Europe Conference Exhibition (DATE '10)*, 2010.

[38] V. Shnayder, M. Hempstead, B. R. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys '04)*, 2004.

[39] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. In *J. ACM*, 29 (4), 1982.

[40] C. Strydis and G. N. Gaydadjiev. Profiling of lossless-compression algorithms for a novel biomedical-implant architecture. In *6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '08)*, 2008.

[41] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '10)*, 2010.

[42] V. Sundaram, P. Eugster, and X. Zhang. Demo abstract: Diagnostic tracing of wireless sensor networks with TinyTracer. In *10th International conference on Information Processing in Sensor Networks (IPSN '11)*, 2011.

[43] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *2nd ACM Conference on Embedded Networked Sensor Systems (SenSys '04)*, 2004.

[44] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *4th Fourth International Symposium on Information Processing in Sensor Networks (IPSN '05)*, 2005.

[45] N. Tsiftes, A. Dunkels, T. Voigt. Efficient sensor network reprogramming through comparison of executable modules. In *5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '08)*, 2008.

[46] T. A. Welch. Technique for high-performance data compression. In *IEEE Computer*, 17 (6), 1984.

[47] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[48] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *5th ACM Conference on Embedded Networked Sensor Systems (SenSys '07)*, 2007.