Compressing Microcontroller Execution Traces to Assist System Analysis

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, Lydie Du Bousquet

Compressing Microcontroller Execution Traces to Assist System Analysis

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie du Bousquet

Université Grenoble Alpes
Laboratoire d'Informatique de Grenoble
38041 Grenoble, France.
FirstName.LastName@imag.fr

Abstract. Recent technological advances have made possible the retrieval of execution traces on microcontrollers. However, the huge amount of data in the collected trace makes the trace analysis extremely difficult and time-consuming. In this paper, by leveraging both cycles and repetitions present in an execution trace, we present an approach which offers a compact and accurate trace compression. This compression may be used during the trace analysis without decompression, notably for identifying repeated cycles or comparing different cycles. The evaluation demonstrates that our approach reaches high compression ratios on microcontroller execution traces.

1 Introduction

A microcontroller is an integrated circuit embedded in various kinds of equipment such as cars, washing machines or toys. Surprisingly, if microcontrollers are now affordable, the development of embedded software is still expensive. According to our industrial partners, this development cost is mainly due to the validation step, and especially debugging. Indeed, though there are several development environments for embedded applications, there exist few tools dedicated to their validation. Consequently, validation and debugging are carried out manually, and thus are tedious and time consuming tasks [12]. Recent microcontrollers allow trace recording. Using specialized probes it is possible to collect basic execution traces without input/output data. Due to the cyclic nature of most embedded programs, such traces consist in very long sequences of multiple repetitions of instructions.

In this paper, we aim to help automated or manual analysis of microcontroller traces by facilating the localization repetitions and by keeping the amount of data manageable. We propose a compression approach based on a grammar generation. Our algorithm, named Cyclitur, is based on our extension of the Sequitur algorithm [11]. Sequitur produces a grammar by leveraging *regularities* found in an input trace. The output grammar is an accurate but compact representation of the input trace. Compared with Sequitur, our extension, named ReSequitur, ensures an additional grammar property. Cyclitur is implemented in a tool named CoMET (see Figure 7). CoMET enables us to compress real traces recorded on embedded applications and network traffic simulations.

```
1  int main(void) {
2    while(1) {
3      static JOY_State_TypeDef JoyState = JOY_NONE;
4      static TS_STATE* TS_State;
5      JoyState = IOE_JoyStickGetState();
6      switch (JoyState) {
7        case JOY_NONE:
8          LCD_DisplayStringLine(Line5, "JOY: ---- "); break;
9        case JOY_UP:
10         LCD_DisplayStringLine(Line5, "JOY: UP "); break;
11       case JOY_DOWN:
12         LCD_DisplayStringLine(Line5, "JOY: DOWN "); break;
13       default:
14         LCD_DisplayStringLine(Line5, "JOY: ERROR "); break;
15     }
16     TS_State = IOE_TS_GetState();
17     Delay(1);
18     if (STM_EVAL_PBGetState(Button_KEY) == 0) {
19       STM_EVAL_LEDToggle(LED1);
20       LCD_DisplayStringLine(Line4, "Pol: KEY Pressed");
21     }
22     if (STM_EVAL_PBGetState(Button_TAMPER) == 0) {
23       STM_EVAL_LEDToggle(LED2);
24       LCD_DisplayStringLine(Line4, "Pol: TAMPER Pressed");
25     } } }
```

```
[...]
0x08000CA0,"  BL.W     LCD_WriteReg (0x08000D74)"
0x08000D74,"  PUSH     {r4-r6,lr}"
0x08000D76,"  MOV      r5,r0"
0x08000D78,"  MOV      r4,r1"
0x08000D7A,"  MOV      r0,r5"
0x08000D7C,"  BL.W     LCD_WriteRegIndex (0x08000DC8)"
0x08000DC8,"  PUSH     {r4,lr}"
0x08000DCA,"  MOV      r4,r0"
0x08000DCC,"  MOV      r0,#0x70"
[...]
```

Fig. 1. Example of C embedded software code and extract from execution trace

2 Motivation

Microcontrollers run software programs specially designed for embedded use. Embedded programs are most often written in the C programming language. A lot of those programs can be categorized as *cyclic*, i.e., they rely on a main loop that iterates indefinitely. In the following, we call the *loop header* the instruction that defines this main loop. Usually, at each iteration of the loop, sensors are read and actions are taken in response. Figure 1 gives a small example of embedded software in C. This program repetitively checks if the user moves a joystick or pushes a button, and displays some text on an LCD screen to describe the actions of the user. The cyclic aspect is represented by an infinite loop, which starts at line 2.

The cost of developing software for microcontrollers is still very high. The specificity of each use case and the very low-level programming render the development of such software error-prone. Consequently, a very large part of the development time is spent in debugging. However, the arrival of new microcontrollers has made possible the recording of execution traces. For instance, ARM Cortex-M microcontrollers include a module dedicated to trace recording, called Embedded Trace Macrocell. Using a specific probe, it is possible to record the execution trace of the program running on the microcontroller. Although race

analysis seems pertinent to debug microcontroller programs, the collected traces usually contain a huge amount of data, due to the real-time and cyclic nature of the embedded programs. Figure 1 provides a small extract of the execution trace collected during the execution of the program. While the recorded trace was more than one million lines long, the chosen extract contains a few lines occurring during a call of the macro `LCD_DisplayStringLine`. This extract exemplifies what makes the trace analysis very difficult: a few lines of source code can give raise to a very large amount of data in the trace.

This paper proposes a method to assist the analysis process of microcontroller execution traces. In other words, we address the first two steps of the debug process, which are the comprehension and the analysis of execution traces. Indeed, we propose a method to compress the microcontroller execution traces [10,8,7], in order to have a high view to get a quick understanding of execution traces. Our tool CoMET (see Figure 7) implements our compression, and provides two visualizations. The engineer, by visualizing the compression generated by our approach, will be guided throughout the trace analysis, for instance, by identifying cycles that appears most often in the trace or by comparing cycles.

3 Cyclic trace compression

Given an input string, a grammar-based compression computes a small grammar that generates only one string, the input string. This grammar reveals the structure of the string and can often be used in further processing with no prior decompression, which is an opportunity for trace analysis.

Sequitur, proposed by Nevill-Manning and Witten [11], is a grammar-based compression algorithm. To generate a grammar, Sequitur takes a string as input, and finds repeated subsequences present in the string. Sequitur operates in linear time and in an online fashion. Each repetition gives rise to a rule in the grammar, and is replaced with a nonterminal symbol. The compression process is executed iteratively. For instance, for the string *cabcabcabcabcad*, Sequitur generates the following grammar:

$$S \to AABd, \quad A \to CC, \quad B \to ca, \quad C \to Bb$$

The original string contains 15 symbols and the Sequitur-generated grammar contains 14 symbols. The compression explicitly capture the repetitions of the subsequence *cba*.

In this paper we propose Cyclitur, an extension of Sequitur, to compress microcontroller execution traces. While keeping the same complexity as Sequitur, Cyclitur compresses consecutive repetitions and takes advantage of the cyclic nature of the trace. For instance, for the same string and if the loop header is a, Cyclitur computes this grammar:

$$S \to cA^4B, \quad A \to abc, \quad B \to ad$$

The grammar contains only 11 symbols and each of the cycles is represented by a single symbol (c as is, abc as A, and ad as B). This section first explains our formalism and details the improvements made to Sequitur to compress microcontroller execution traces.

3.1 Preliminaries

Given an alphabet Σ, an *r-string* α is a sequence of pairs $\langle$*symbol*, *number of consecutive repetitions*$\rangle$. The set of *r-strings* over Σ is $\Sigma_r^* = (\Sigma \times \mathbb{N} \setminus \{0\})^*$. In an r-string α, $\alpha_{i,1}$ stands for the symbol of the i-th element of α and $\alpha_{i,2}$ for its number of repetitions. $|\alpha|$ denotes the number of elements in the r-string α. To lighten notations, repetition numbers are placed in superscript after symbols and they are omitted when equal to one. For instance, ab^5d^{10} is a shorthand for the sequence $\langle a,1\rangle\langle b,5\rangle\langle d,10\rangle$. The *expansion* of the r-string $\alpha \in \Sigma_r^*$, noted $\widehat{\alpha}$, is a string in Σ^*, and is defined as follows:

$$\widehat{\alpha} = \underbrace{\alpha_{1,1} \cdots \alpha_{1,1}}_{\text{repeated } \alpha_{1,2} \text{ times}} \underbrace{\alpha_{2,1} \cdots \alpha_{2,1}}_{\alpha_{2,2} \text{ times}} \cdots \underbrace{\alpha_{|\alpha|,1} \cdots \alpha_{|\alpha|,1}}_{\alpha_{|\alpha|,2} \text{ times}} .$$

An *r-grammar* G is a 4-tuple $\langle \Sigma, \Gamma, S, \Delta \rangle$ where:
- Σ is a finite alphabet of *terminal* symbols,
- Γ is a disjoint finite alphabet of *nonterminal* symbols,
- $S \in \Gamma$ is a *start symbol*, i.e., a particular nonterminal,
- and $\Delta \subseteq \Gamma \times (\Sigma \cup \Gamma)_r^*$ is a set of *r-production rules*,

such that the following properties are verified:
- for every nonterminal A, there is a unique r-string α s.t. $\langle A, \alpha\rangle$ is in Δ,
- there is an ordering over Γ s.t., for each r-production rule $\langle A, \alpha\rangle$ in Δ, every nonterminal in α precedes A.

An r-production rule $\langle A, \alpha\rangle \in \Delta$ associates the nonterminal A and the r-string α, resp. called the *head* and the *body* of the rule. The *r-grammar body* is $\{\alpha \mid \exists A : \langle A, \alpha\rangle \in \Delta\}$, i.e., the set of rule bodies. Note that the additional properties ensure that an r-grammar contains one rule per nonterminal and is non recursive (cycle-free). In the following, we consider an r-grammar $G = \langle \Sigma, \Gamma, S, \Delta\rangle$.

3.2 Properties of generated grammars

ReSequitur is an algorithm that compresses a string to an r-grammar (see Figure 2). ReSequitur takes as inputs an alphabet Σ, a string to compress $\omega \in \Sigma^*$, a (possibly empty) initial set of symbols Γ_0, and an initial set of rules $\Delta_0 \subset (\Gamma_0 \times \Sigma_r^*)$. Like Sequitur, ReSequitur ensures that two properties called *digram uniqueness* and *rule utility* hold on the output r-grammar.

The digram uniqueness property states that an r-grammar should not contain two non-overlapping occurrences of the same digram in the r-grammar body.

Property 1 (Digram uniqueness)**.** *The digram uniqueness property holds for G, noted $RUniqueness(G)$, if for all terminals $A, B \in \Gamma$, symbols $a, b, c, d \in \Sigma \cup \Gamma$, strictly positive integers $n, m, p, q \in \mathbb{N} \setminus \{0\}$, and r-strings $\alpha, \beta, \gamma, \delta \in (\Sigma \cup \Gamma)_r^*$, the two following statements hold:*

$$(A \neq B \wedge \{\langle A, \alpha\, a^n b^m\, \beta\rangle, \langle B, \gamma\, c^p d^q\, \delta\rangle\} \subset \Delta) \implies a^n b^m \neq c^p d^q \qquad \text{(in different rules)}$$

$$(\langle A, \alpha\, a^n b^m\, \beta\, c^p d^q\, \gamma\rangle \in \Delta) \implies a^n b^m \neq c^p d^q \qquad \text{(in a same rule)}$$

The rule utility property ensures that every rule except the start rule is used more than once in the r-grammar body. This is formally defined as follows:

```
let S be a fresh nonterminal representing a rule (S ∉ Σ ∪ Γ_0)
G ← ⟨Σ, Γ_0 ∪ {S}, S, Δ_0 ∪ {⟨S, ϵ⟩}⟩
for i ← 1 to |ω| do
    append (ω_i)^1 to the body of rule S
    while ¬RUniqueness(G) ∨ ¬RUtility(G) ∨ ¬RConsecutive(G) do
        if ¬RConsecutive(G) then
            let a, n, m be s.t. a^n a^m is a r-digram in G
            replace every occurrence of a^n a^m in G with a^{n+m}
        else if ¬RUniqueness(G) then
            let δ be a repeated r-digram in G
            if ∃⟨A, α⟩ ∈ Δ : α = δ then
                replace the other occurrence of δ in G with A
            else
                form new rule ⟨D, δ⟩ where D ∉ (Σ ∪ Γ)
                replace both occurrences of δ in G with D
                Δ ← Δ ∪ {⟨D, δ⟩}
                Γ ← Γ ∪ {D}
            end
        else if ¬RUtility(G) then
            let ⟨A, α⟩ ∈ Δ be a rule used once
            replace the occurrence of A with α in G
            Δ ← Δ ∖ {⟨A, α⟩}
            Γ ← Γ ∖ {A}
        end
    end
end
return G
```

Fig. 2. Function ReSequitur$(\Sigma, \omega, \Gamma_0, \Delta_0)$

Property 2 (Rule utility)**.** *The rule utility holds for G, noted $RUtility(G)$, if:*

$$\forall A \in \Gamma \setminus \{S\} : \left(\sum_{\langle B,\beta\rangle \in \Delta} \sum_{i \in [1..|\beta|]} \begin{cases} \beta_{i,2} & \text{if } \beta_{i,1} = A \\ 0 & \text{if } \beta_{i,1} \neq A \end{cases} \right) \geq 2 \,.$$

In order to compress consecutive repetitions more efficiently, compared with Sequitur, ReSequitur ensures an additional property: any digram in an r-grammar body consists of different symbols.

Property 3 (No consecutive repetition)**.** *G has* no consecutive repetitions*, which is noted $RConsecutive(G)$, if the following statement holds:*

$$\forall a, b \in \Sigma \cup \Gamma, \forall n, m \in \mathbb{N} \setminus \{0\}, \forall \alpha, \beta \in (\Sigma \cup \Gamma)_r^*, \forall C \in \Gamma :$$
$$\langle C, \alpha\, a^n b^m \beta\rangle \in \Delta \Rightarrow a \neq b \,.$$

To ensure this property at each iteration of the outer loop, ReSequitur merges every digram of the form $a^n a^m$ into a single repeated symbol a^{n+m}.

3.3 Exploiting cycles with Cyclitur

Recall that our objective is to compress cyclic traces extracted from microcontrollers. Therefore, the first step in our approach is the *cycle detection*. A cycle is

```
1 Γ0 ← ∅;  Δ0 ← ∅;  ω′ ← ϵ;  i ← 1
2 for  j ← 2 to |ω| do
3 |   if j = |ω| ∨ ωj = lh then
4 |   |   ⟨Σ′, Γ′, S′, Δ′⟩ ← ReSequitur(Σ, ωi..j−1, Γ0, Δ0)
5 |   |   Γ0 ← Γ′;  Δ0 ← Δ′;  ω′ ← ω′ · S′;  i ← j
6 |   end
7 end
8 ⟨Σ″, Γ″, S″, Δ″⟩ ← ReSequitur(Σ, ω′, Γ0, Δ0)
9 return ⟨Σ″, Γ″, S″, Δ″⟩
```

Fig. 3. Function Cyclitur(Σ, ω, lh)

a subsequence of the execution trace that consists of one execution of the main loop of the embedded program. Cycle detection relies on the localization of a special event that represents the loop header. Detecting cycles using the loop header event consists in dividing the trace into blocks, where each block represents a specific cycle. Given a loop header lh, consider the set of cycles $C(\omega, lh)$ defined as a set of pairs of indexes over the execution trace ω as follows:

$$C(\omega, lh) = \{\langle i, j\rangle \in [1..|\omega|]^2 \mid i \leq j \wedge (i = 1 \vee \omega_i = lh) \wedge (j = |\omega| \vee \omega_{j+1} = lh) \\ \wedge \forall k \in [i+1..j] : \omega_k \neq lh\}\,.$$

Figure 3 presents the overall algorithm of the Cyclitur compression. ReSequitur is first applied on each cycle to detect repetitions (lines 2–7), while sharing the same set of rules. Then applying ReSequitur on the compression produced by the previous step allows to detect similar sequences of cycles in the trace (line 8).

4 Application example

Cyclitur infers patterns in execution traces and can be used for many applications. As an example of such an application, we propose here to use the compression generated by Cyclitur to detect an abnormal behavior in the embedded context. As shown in Figure 4, in this example, a user equipped with a device (e.g. a smartphone) interacts with five sensors. The user turns from the right to the left, and whenever he is in front of a sensor, his device sends a message to the sensor. After receiving a notification from the device, the sensor sends a message to the user. Finally, when the user receives the message sent by the sensor, his device sends an acknowledgment to the sensor. If the user sends a message to a sensor, and he does not get a response from the sensor, within 15 seconds, he considers this behavior as abnormal behavior and he turns to the left and interacts with the next sensor. Figure 5 illustrates the execution trace generated using this example.

By using the word "*send*" as the loop header in the trace, the execution trace will be divided into blocks as shown in Figure 5. For example, the events that are related to the sending of a message from the device to a sensor will be compressed and represented by $C1$. Then the compression generated by Cyclitur

will be of the form:

$$S \rightarrow A^{(n\ times)}, \quad A \rightarrow C1\ C2\ C3$$

Let us consider the following senario, the sensor 3 receives a message from the user, but it takes more than 15 seconds to answer. In this case, the user turns to the left and interacts with the sensor 4. While the user receives the message from the sensor 4, his device receives also the message from the sensor 3. Therefore, the device of the user sends an acknowledgment to sensor 3 and an acknowledgment to sensor 4. Using the trace compression illustrated in Figure 6, which is generated by Cyclitur and where the normal behavior is $C1\ C2\ C3$, it is intuitive to note that the abnormal behavior is $C1\ C1\ C2\ C2\ C3\ C3$.

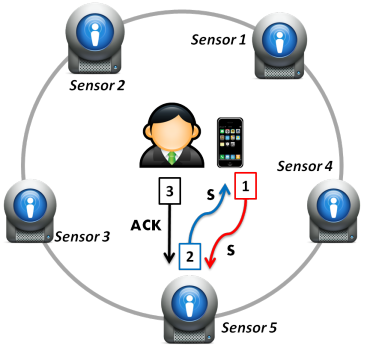

Fig. 4. Application example

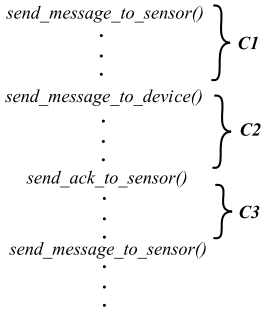

Fig. 5. Execution trace of the application example

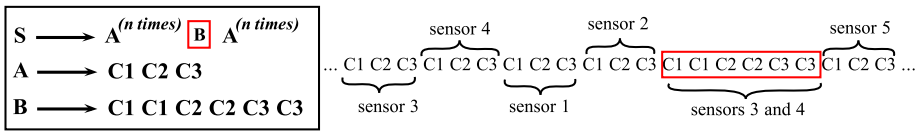

Fig. 6. Anomaly detection using trace compression

5 Implementation and Evaluation

The evaluation of our trace-compression approach is an experimental evaluation that consists in comparing grammars obtained by applying Sequitur and Cyclitur on various execution traces. The experimental evaluation was made possible thanks to our tool named CoMET.

5.1 CoMET

CoMET is a tool written in Java in 12,000 LOC that implements both Sequitur and Cyclitur algorithms. It takes as input an execution trace file. It extracts automatically a string of symbols, to finally output a grammar, either as text or as a Java object for programmatic use. As shown in Figure 7, CoMET provides two visualizations:

- *Cycle pie chart*: it provides the occurrence rates of cycles in the trace.
- *Cycle in time*: it emphasizes all occurrences of a specific cycle in the trace.

5.2 Metrics of experimental evaluation

In the following, we use a given string (trace) ω, and the output grammar (resp. r-grammar) generated with Sequitur (resp. Cyclitur), noted $G = \langle \Sigma, \Gamma, S, \Delta \rangle$. The *size* of the grammar G is the sum of the number of symbol occurrences in its body (both terminals and nonterminals) and the number of its rules. The *compression ratio*, noted $Comp(G)$, is used to compare the degree of compression of grammars generated using Sequitur and Cyclitur and is defined as follows:

$$Comp(G) = \frac{Size(G)}{|\omega|}$$

Note that the compression ratio varies between 0 and 1, where 0 represents the best compression, and 1 the worst.

5.3 Programs and traces

The traces used to evaluate our approach come from five embedded programs provided by STMicroelectronics and EASii IC. For confidentiality reasons, programs are not described. In the following we denote by Pi the i-th program. One at a time each program is loaded onto a STM32F107 EVAL-C microcontroller board and executed. The execution trace is recovered using a Keil UlinkPro probe, and saved in *CSV* format. For each program, five execution traces are produced. In the trace file, for each instruction, we have the *time* when it was executed, the corresponding *assembly instruction* and the *program counter (PC)*. For our compression approach we are only interested in the PCs.

Table 1. Evaluation results

Prog.	**Trace**	**#Sym.**	**#Cycles**	**Sequitur**		**Cyclitur**	
				$Size(G)$	$Comp(G)$	$Size(G')$	$Comp(G')$
P1	T1	1048575	7588	2581	0.002461436	1484	0.001415254
	T2	1048576	7274	1748	0.001667023	944	0.000900269
	T3	1048576	7109	1764	0.001682281	946	0.000902176
	T4	1048571	7586	1497	0.001427657	970	0.000925068
	T5	1048574	4448	1051	0.001002314	625	0.000596048
P2	T1	1048575	1515	21040	0.020160694	18102	0.017263429
	T2	1048575	1593	20324	0.019382495	16537	0.015770927
	T3	1048575	1591	18933	0.018055933	15970	0.015230193
	T4	1048576	1736	19658	0.01874733	16709	0.015934944
	T5	1048574	1789	19478	0.018575704	17335	0.016531976
P3	T1	1048572	1440	1918	0.001829154	1766	0.001684195
	T2	1048571	1442	1830	0.001745232	1407	0.001341826
	T3	1048573	1442	1813	0.001729016	1686	0.001607899
	T4	1048576	1441	1961	0.001870155	1661	0.001584053
	T5	1048576	1442	1842	0.001756668	1701	0.0016222
P4	T1	1048567	1277	1726	0.001646056	1407	0.001341831
	T2	1048571	1462	2199	0.00209714	1488	0.001419074
	T3	1048574	1276	1879	0.001791957	1325	0.001263621
	T4	1048575	1462	1706	0.00162697	1524	0.001453401
	T5	1048575	1462	1613	0.001538278	1416	0.001350404
P5	T1	1048573	16132	301	0.000287057	125	0.00011921
	T2	1048576	16132	295	0.000281334	132	0.000117302
	T3	1048570	7440	1599	0.001524934	1172	0.001117713
	T4	1048571	16131	309	0.000294687	137	0.000130654
	T5	1048576	16132	290	0.000276566	135	0.000128746

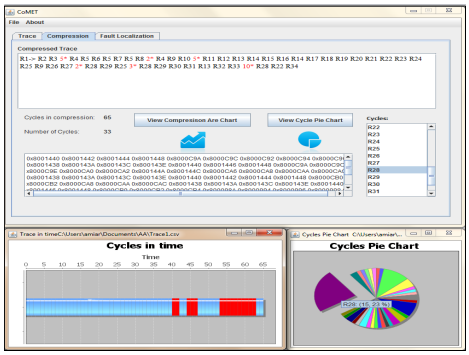

Fig. 7. CoMET Visualization

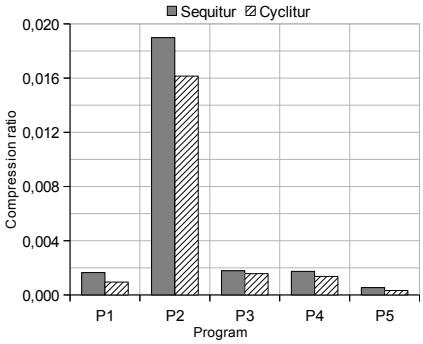

Fig. 8. Compared average compression ratios for each program

5.4 Results

Table 1 contains the results of the experimental evaluation, where each line represents a trace of a program. The columns **#Sym.** and **#Cycles** represent respectively the number of symbol occurrences and the number of cycles in a trace. For each generated grammar, Table 1 contains its size *Size*() and its compression ratio *Comp*(). Figure 8 displays the arithmetic average of the compression ratios over the five collected traces of the program. Note that the use of other average measures give different values, but the same result: a clear difference of compression ratios between Sequitur and Cyclitur. The compression ratio varies between 0 and 1, where 0 represents the best compression, and 1 the worst. For example, for program P1, we observe that our approach produces better compression than Sequitur. The sizes of grammars generated by Sequitur for the execution traces of P1 vary from 1,051 up to 2,581. The sizes of original traces vary between 1,048,571 and 1,048,576, with 4,448 and 7,588 cycles. Therefore, the compression ratios vary between 0.0010 and 0.0025. Cyclitur produces grammars whose sizes vary from 625 to 1,484. The compression ratios vary between 0.0006 and 0.0014. Note that, for all considered programs, the use of Cyclitur leads to better compression than Sequitur. The compression ratios are better from 12% to 42%.

5.5 Cyclitur and network traces

We evaluate our trace-compression approach on four additional traces obtained from network simulations. The considered network is a Multi-Channel Multi-Interface Wireless Mesh Network (WMN) with routers based on the IEEE 802.11 technology [4]. The loop header used to detect cycles is a specific event that refers to the emission of a request from client to server.

The *trace 1* consists in 6,011,850 events spread over 9,574 cycles. The compression ratio using Sequitur is 0.0027% for a generated grammar whose size is 16,531 (744 terminals, 12,375 nonterminals and 3,412 rules). The compression ratio for Cyclitur grammar reaches 0.0026% for a grammar of size 16,057 (373 terminals, 13,884 nonterminals and 4,182 rules). Cyclitur generates a grammar that contains more rules than the grammar generated by Sequitur, but is

easier to understand and to analyze, because it is more compact and it facilitates cycle detection. The *trace 2* trace consists in 8,040,942 events spread over 9,574 cycles. Sequitur generates a grammar of size 18,639 while Cyclitur generates a grammar of size 18,439. The use of Sequitur and Cyclitur on the second trace gives respectively 0.00023% and 0.0022% compression ratios. The *trace 3* contains 13,883,977 events spread over 2,797 cycles. While Sequitur generates a grammar of size 28,181, Cyclitur generates a grammar of size 26,468. The compression ratios are respectively 0.0020% and 0.0019%. Finally, the *trace 4* contains 10,312,955 events spread over 33,834 cycles. While Sequitur reaches a 0.0016% compression ratio with a grammar of size 16,690. Cyclitur reaches a 0.0009% compression ratio with a grammar of size 10,145.

The previous results show that Cyclitur can be used likewise to compress network traffic traces. We observe that for all network traces collected in these experiments, the use of Cyclitur generates a better compression than Sequitur.

6 Related Work

Compressing microcontroller traces with the objective of analyzing them remains a challenge. In other areas, particularly in object oriented context, there are numerous studies concerning reduction and compression of execution traces. Hamou-Lhadj and Lethbridge [8] use an acyclic oriented graph representing method calls to compress traces. Other representations have been used such as trees [13] and finite automata [9]. Also, in [7] Hamou-Lhadj and Lethbridge, propose the removal of implementation and useless details to ease the analysis of execution traces. These object oriented approaches are not suitable for our purpose for multiple reasons. First, they discard the order of events, which is paramount to understand a program. Second, they use input/output data. In our context, this information is rarely available and raises important storage problems. Third, they reason about method calls. In optimized microcontroller code, function calls alone are inadequate to understand the program, since the core logic of a program is sometimes coded in a single function.

Generic data compression methods have been used for execution traces, e.g., Gzip [6]. However, almost no analysis can be performed on the compressed form of the execution trace. Larus [10] proposes to compress control flow paths using Sequitur. As Sequitur finds regularities in the path (e.g., repeated code), the output grammar can be used to detect hot subpaths, i.e., short acyclic paths that are costly. Our work consists in compressing traces by detecting and exploiting cycles. It allows us to reach better compression ratios than Sequitur. Burtscher et al. [2] also propose a value predictor-based compression algorithm for execution trace that obtain better compression ratios than Gzip or Sequitur. Zhang and Gupta [15] propose an improvement of value predictor-based compression for whole execution trace (WET), i.e., control flow and other information mixed together. Their method allows the user to extract partial information (e.g., the control flow path) from the compressed WET. If their method allows bidirectional decompression, such compression are usually not understandable by the

engineer and automated analysis requires at least partial decompression. On the contrary, the simple structure offered by the grammar is well-suited for analysis. Moreover, these techniques may notably lose reference points (here, cycles). Like the run-length encoding, Cyclitur compresses consecutive repetitions but it also detects patterns and cycles. We believe that cycles revealed in the process of Cyclitur may assist automatic trace analysis, e.g., using cycle matching [1].

Grammar-based compression is the object of active research in information theory (cf. [3] for a survey). In particular, extensions of Sequitur are able to produce smaller grammars. For instance, Yang and Kiefer propose to generalize Sequitur to n-grams (rather than digram) [14]. If this leads to better compressions, it comes at a price: their algorithm does not share the time and space complexity of Sequitur, and as a result, is not usable on large amount of data.

7 Conclusion and Perspectives

In the microcontroller context, new microprocessors have enabled the recording of the execution of embedded software as a trace. Analyzing execution traces may help in embedded software debugging, which represents a large part of the cost of their development. However, collected traces contains a huge amount of data, making the analysis difficult and tedious. For both manual and automatic analysis of the trace, it seems opportune to have a compact and analyzable representation of the trace.

In this paper, we propose a trace compression method that aims at facilitating trace analysis. The method relies on a grammar-based compression named *Cyclitur* built upon the Sequitur algorithm [11]. Our approach starts by dividing a trace into cycles, where each cycle is an execution of the active main loop. The second step consists in discovering and compressing similarities in the trace.

Our approach is evaluated to compare its compression rate to the existing Sequitur algorithm. The experimental evaluation shows that our approach generates an equivalent or better compression than Sequitur on execution traces. On microcontroller execution traces, Cyclitur compression ratios were better than Sequitur compression ratios from 12% to 42%. In addition, Cyclitur may help in identifying and locating important details in an execution trace.

While Cyclitur is not aimed at competing with compression ratio of compression schemes in general, it would be interesting to compare Cyclitur with value predictor-based compressions [15,2]. Also, we intend to help locating faults in embedded software by analyzing compressed traces and adapting dynamic validation and data mining techniques [5].

Acknowledgment

This work has been funded by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project (Instrumentation and Tools for 32-bit Microcontrollers). The authors would like to thank STMicroelectronics, AIM and ESAii IC for their help.

References

1. J. Aoe. *Computer Algorithms: String Pattern Matching Strategies*. Wiley-IEEE Computer Society Press, 1994.
2. M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam. The VPC trace-compression algorithms. *IEEE Trans. on Computers*, 54(11):1329–1344, Nov. 2005.
3. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. on Information Theory*, 51(7):2554–2576, July 2005.
4. C. De Oliveira, F. Theoleyre, and A. Duda. Connectivity in multi-channel multi-interface wireless mesh networks. In *International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 35–40, 2011.
5. U. Fayyad, G. Piatetsky-shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.
6. J.-L. Gailly and M. Adler. Gzip. `http://www.gzip.org`.
7. A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *International Conference on Program Comprehension (ICPC)*, pages 181–190. IEEE Computer Society, 2006.
8. A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *International Workshop on Program Comprehension (IWPC)*, pages 159–. IEEE Computer Society, 2002.
9. M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *International Symposium on Static Analysis (SAS)*, pages 69–85. Springer-Verlag, 2009.
10. J. R. Larus. Whole program paths. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 259–269. ACM, 1999.
11. C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *Journal of Artificial Intellgence Research (JAIR)*, 7:67–82, 1997.
12. A. Rohani and H. Zarandi. An analysis of fault effects and propagations in AVR microcontroller ATmega103(L). In *International Conference on Availability, Reliability and Security (ARES)*, pages 166–172, 2009.
13. K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of Java program. In *International Workshop on Principles of Software Evolution*, pages 148–154. IEEE Computer Society, 2005.
14. E.-H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform—Part one: Without context models. *IEEE Trans. on Information Theory*, 46(3):755–777, May 2000.
15. X. Zhang and R. Gupta. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.*, 2(3):301–334, Sept. 2005.