Software Diagnosis using Compressed Signature Sequences

István MAJZIK

Technical University of Budapest, Hungary
Department of Measurement and Instrument Engineering
H-1521 Budapest, Müegyetem rkp. 9.
E-mail: majzik@mmt.bme.hu

Abstract: Software diagnosis can be effectively supported by one of the concurrent error detection methods, the application of watchdog processors (WP). A WP, as a coprocessor, receives and evaluates signatures assigned to the states of the program execution. After the checking, the watchdog stores the run-time sequence of signatures which identify the statements of the program. In this way, a trace of the statements executed before the error is available. The signature buffer can be effectively utilized if the signature sequence is compressed. In the paper, two real-time compression methods are presented and compared. The general method uses predefined dictionaries, while the other one utilizes the structural information encoded in the signatures.

1 Introduction

Post-mortem diagnosis of embedded (real-time) application programs is often difficult, since in most of the cases diagnosis is supported only by static information in some form of a memory dump. The system designer would rather be interested in the trace of the erroneous program, i.e. in the sequence of statements executed by the program before the error. To take and store this trace, often complex and costly monitoring systems should be implemented which may even modify the original operating environment and interfere with the timing of the monitored programs. However, it has to be pointed out that the error detection mechanisms implemented in highly dependable systems often provide mechanisms to derive the trace of the program with minor cost and additional effort. Our goal is to show, that one of the commonly used run-time error detection methods, the application of watchdog-processors, can be easily extended to support the trace based post-mortem diagnosis of the program.

Dependable applications require continuous, concurrent run-time error detection mechanisms in order to highlight transient errors causing disturbances in data and control flow. Errors in data can be effectively detected (and even corrected) by error detecting and correcting codes, while one of the most efficient methods for the detection of control-flow errors is the appli-

cation of watchdog-processors. A watchdog-processor (WP [1]) is a relatively simple coprocessor monitoring the state of the system using signatures, compact abstractions of the system state. In the assigned signatures method ([2]), the checked program is modified at compilation time by a preprocessor in such a way, that during the run the signatures are transferred to the WP. (The preprocessor analyzes the high-level program text, labels the statements of the program by signatures and inserts the signature transfer instructions.) The WP evaluates the run-time sequence of signatures on the basis of a predefined reference. If a signature is found which is not a valid successor of the previous signature then a control-flow error is detected: the program entered an erroneous state inconsistently with the control flow graph.

The signatures assigned by the preprocessor uniquely identify the states of the program. In the default case, each individual statement of the program is associated with an unique signature, but additional reduction phases can merge branch-free statement sequences into a block labelled by a single, joint signature. In this way, the run-time sequence of signatures contains the necessary information on which basis the execution of the program, the trace of the statements can be later restored. However, the original error detection mechanism does not store this sequence of run-time signatures in the WP. If a signature is accepted as a valid one, then the next reference value is derived and the actual signature is deleted, the WP is prepared to receive and evaluate the next signature.

If the run-time signature sequence is stored in the WP, then a complete log of the program execution is available, the trace of executed statements can be restored. The difficulty is that for practical programs this sequence is too long to store in full extent. Iteration loops, frequently called procedures, synchronization cycles waiting for external events transfer a large number of signatures to the WP preventing the storing of the entire sequence in a buffer of limited size. A trade-off between the efficiency and moderate cost is to implement a logic analyzer-like cyclic buffer storing a limited log of signatures transferred before an error was detected. The utilization of the cyclic buffer can be further improved by some kind of information compression on the signature sequence before storing the log, since the majority of signatures originates from repetitive signature sub-sequences.

The basic idea is summarized as follows: The WP receives and compresses the run-time sequence of signatures. The compressed sequence is stored in a cyclic buffer. If an error is detected then the application is stopped and the buffer can be read by the diagnosis program. The content of the buffer is decompressed, the original signature sequence and the statements identified by the signatures are derived. In this way the trace of the statements executed before the error is available for diagnosis purposes.

In Section 2 the compression of a general signature sequence is investigated. A dictionary is constructed on the basis of the control flow graph (CFG) of the program to be monitored. It contains the necessary program-specific information in order to ensure the optimal compression of the run-time signature sequence.
The special structure of the signatures used in the *Signature Encoded Instruction Stream* (SEIS [3]) signature assignment method allows the definition of a general compression scheme. In this case no dictionary is needed, the run-time signature sequence can be com-

pressed without downloading or building any program-specific dictionary. This interesting (sub-optimal) compression is discussed in Section 3. At the end, measurement results (Section 4) and the proposed diagnostic environment (Section 5) are presented.

2 Compression of the signature sequence using a predefined dictionary

The theoretical problem of the compression of the signature sequence is a problem of universal encoding. In our case, the message is the run-time sequence of signatures, the message alphabet contains the valid signatures while the encoding alphabet consists of a fixed number of *characters*. The signature sequence is divided into *words* (sets of subsequent signatures) of varying size and each word is encoded by a single character of the encoding alphabet. The words and the corresponding characters form a *dictionary.*

In the common universal encoding schemes (Adaptive Huffman, Lempel-Ziv I-II) the dictionary is built in run-time. The run-time construction of the dictionary is time-consuming and needs a fast, difficult and sophisticated hardware. We propose a method which allows a simpler hardware compressor unit by using a predefined dictionary. The dictionary is built during compilation, when the program is analyzed, the CFG is derived and the signatures are assigned to the states of the program. Before the start of the program, the dictionary is downloaded into the WP. The compression mechanism operates on the basis of the predefined dictionary: if a word is found in the signature sequence then the corresponding character is stored in the buffer (in the case of repeating characters only a counter is increased). This approach ensures the simplicity of the compression hardware, however, the efficiency of the compression depends on the definition of the dictionary, i.e. on the optimal selection of the words.

The signature sequence is known completely only in run-time due to the data dependency of the program run. But, since the control flow graph of the program is known, program paths assumed to be executed multiple times (at a high rate) can be identified. In this way, signature sequences originating from the execution of these program paths should define the words of the dictionary. The execution of the following structures can be taken into account:

- body of an iteration (loop);
- long (branch-free) sequences of instructions;
- normal branches of selections (exception should rarely occur);
- frequently called small procedures.

The preprocessor which analyzes the program text can identify these structures, derive the signature sequences associated with them and in such a way define the dictionary.

To compress a run-time signature sequence, an additional hardware unit (*signature compressor*) is intended to be built in the WP. It receives the signatures, performs an on-line compression and stores the compacted sequence in the WP-internal *compression buffer.* In the following, first the structure of the predefined dictionary is given, then the compression algorithm and its properties are presented.

2.1 The dictionary

The words of the dictionary are associated with frequently executed program paths, but do not cover all of the possible paths of the program execution. The following considerations ensure that signature sequences, belonging to paths not mapped directly to words, are stored in the compression buffer as well:

- each valid (single) signature is encoded by a unique character;
- prefixes of the defined words are encoded by unique characters as well.

The structure of the dictionary is a *set of trees* representing words starting with a given signature. Each valid signature is a root of an individual tree, followed by its immediate successor signatures according to the CFG and so on until the endpoints of the tree. In this way a node of the tree identifies an unique signature sequence starting with the root signature and ending at the given node.
The nodes of the trees - and, consequently, the signature sequences embedded in the signature tree - are associated with *unique* characters of the encoding alphabet. The character associated with a node encodes the signature sequence along the path from the root of the tree to the given node (Figure 1). The above mentioned requirements are satisfied: each signature, as a root of a tree, is encoded by a unique character, and prefixes of words are encoded by a single character as well.

The construction of the signature trees is optimized in the sense that a postfix of a word is represented by a path in the tree starting with the first signature of the postfix only if it is awaited to occur separately in the signature sequence, not following its prefix in the original word. (E.g. if the sequence of signatures *s1-s2-s4-s7-s8* is a path in the tree starting with *s1* and *s4* always follows *s2*, then the postfix s*4-s7-s8* is not represented by a path in the tree starting with *s4*.)

Accordingly, most of the trees consists of only the root signature, but there are some signatures which are represented by nodes in several signature trees (they are embedded in words starting with different signatures). An additional reason of the repetition of signatures is that the implementation of the compression algorithm requires the limitation of the number of branches at a given node of the tree. (The structural properties of the control flow graphs of programs enable this limitation as most of the signatures have only a limited number of valid immediate successors, the practical limits are 2 or 3.) In Figure 1 an example CFG and the corresponding signature trees are given. The possible paths of the iteration are separately encoded since they are expected to be executed multiple times. Two complete paths are also covered by words.

2.2 The compression algorithm

The task of the compression algorithm is to find the longest word which is encoded by a single character. (In worst case, each signature is encoded by a separate character which corresponds to the root of the tree associated with the signature.) If the maximal word is found then the encoding character is stored into the compression buffer. The compression buffer is a linear array of elements, each element consists of two fields: a field storing the character and a second field counting the subsequent occurrences of the character.

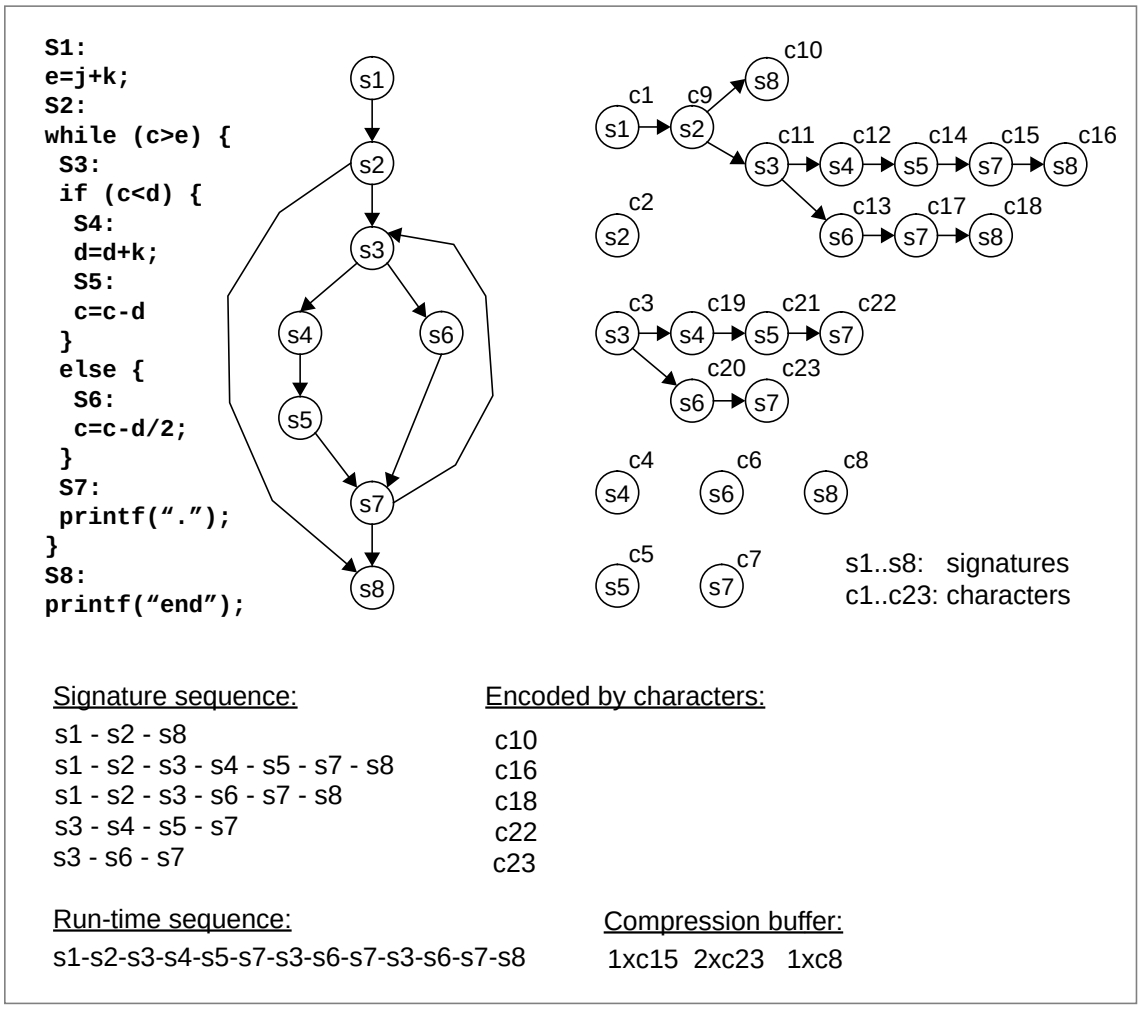

Figure 1 A program CFG and the corresponding signature trees

The compression algorithm starts in the *Start phase* then continues in the *Encoding phase* (signatures are received and processed looking for the most feasible character which encodes the sequence). The character encoding the maximal word is stored in the *Storing phase*.

1 ***Start phase****:* If the first signature of a word has been received then the tree associated with this signature is selected. The actual node is the root node, the next signature is processed in the *Encoding phase*.

2 ***Encoding phase****:* As the next signature is received, the successors of the actual node (which corresponds to the previous signature) are addressed and compared with the actual signature.
If one of them equals to the actual signature then the node corresponding to it becomes the new actual node. The encoding of the word continues in the *Encoding phase*.
If none of them equals to the actual signature (or there are no successors in the tree) then the actual word is ended. The character associated with the actual node is stored into the compression buffer *(Storing phase*), the actual signature is processed in the *Start phase*.

3 ***Storing phase****:* If a word of maximum length (which is encoded by a single character) is found then the character associated with the actual node is stored.
If the actual character is the same as the previous character stored in the buffer then only

its counter has to be increased by one. Otherwise the actual character has to be stored in the next element of the compression buffer (with *1* as counter value).
The actual signature is processed as first signature of the next word (*Start phase*).

2.3 The enhanced compression algorithm

There are (longer) paths in the CFG which share common subpaths. Accordingly, the signature sequences corresponding to these subpaths should be embedded in various longer words. In order to reduce the size of the dictionary, the *enhanced compression algorithm* enables the use of *embedded characters*. If a signature sequence is encoded by a character then instead of the sequence the *character* can be placed into the dictionary. To keep the compression algorithm as simple as possible, only those characters should be used as embedded characters which represent a path (signature sequence) from the root to the end of a signature tree.

Compared with the previous subsection, the signature trees are modified since characters can be encoded by another characters if they are embedded in the signature sequence. The signature trees depicted in Figure 2 illustrate how the size of the original dictionary is reduced.

Similarly to the previous algorithm, the compression begins in the *Start phase* then continues in the *Encoding phase*. If a character is found that should be stored in the compression buffer then the *Storing phase* is called. An additional *character stack* is maintained by the algorithm (storing the predecessors of the embedded characters).

1 ***Start phase:*** If the first signature of a word has been received then the tree associated with this signature is selected. The actual node is the root of this tree. The next signature can be received and processed in the *Encoding phase*.

2 ***Encoding phase:*** The successors of the actual node (which usually corresponds to the previous signature) are addressed.

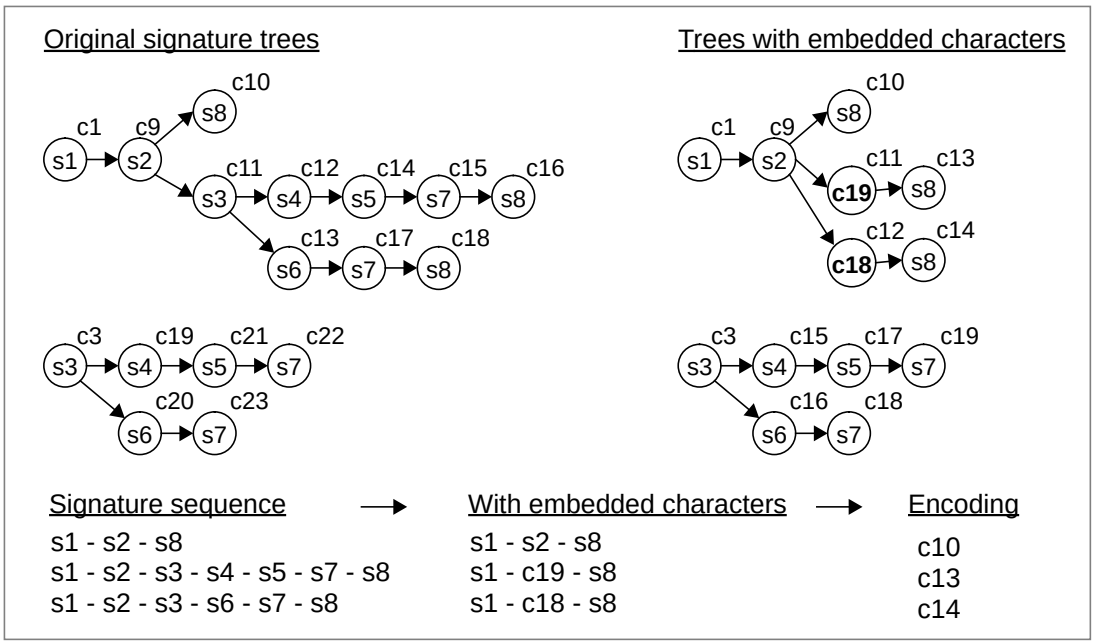

Figure 2 Signature trees with embedded characters

- If there are successors (signatures or characters) of the actual node then first the signature successors are read and compared with the actual signature:
  - If there is a signature successor which equals to the actual signature then the node corresponding to it becomes the new actual node. The next signature is received and the encoding of the word continues in the *Encoding phase*.
  - If none of the signature successors equals to the actual signature (or, there are no signature successors at all) then
    - if there is no character successor of the actual node, then the word is ended, the actual character is stored in the compression buffer (*Storing phase*) and the actual signature is processed by the *Start phase* (as a first signature of a new word).
    - if there is a character successor of the actual node then an embedded word may follow. The actual node is stored on the *character stack*, the actual signature is processed by the *Start phase* (looking for the word belonging to the embedded character).
- If there are no successors (signatures or characters) of the actual node, then the actual word is ended. It has to be examined whether it is an embedded word or not.
  - If there is no node stored on the character stack then the actual word is an individual word. The actual character is stored in the compression buffer (*Storing phase*), the actual signature is processed by the *Start phase* (as a first signature of a new word).
  - If there is a node stored on the character stack, then the actual word (represented by the actual character) has to be examined whether it is the embedded word following the node on the stack. The node stored on the character stack (the node before the embedded character) becomes the actual node. The successor characters of the actual node are addressed and compared with the actual character.
    - If there is a character successor of the actual node which equals to the actual character then the embedded word has been found. The node corresponding to the valid character successor becomes the new actual node, the top of the character stack is deleted, and the actual signature is processed by the *Encoding phase* (continuing the encoding of the word).
    - If there is no character successor of the actual node which equals to the actual character then the actual word is not the embedded word: the original word is ended and additionally a new word is found.
      The characters of the character stack have to be stored in the compression buffer (in the order they were written onto the stack, *Storing phase*), thereafter the actual character has to be stored in the compression buffer as well (*Storing phase*); the actual signature is processed by the *Start phase* (as a first signature of a new word).

3 ***Storing phase:*** If a character is found that encodes a maximum length of signature sequence then it is stored in the compression buffer as follows:
If the actual character is the same as the previous character stored in the compression

buffer then only its counter has to be increased by one. Otherwise the actual character has to be stored in the next element of the compression buffer (using *1* as counter value).

2.4 Implementation of the dictionary

For the sake of effectiveness and high speed of the compressor hardware, the signature trees of the dictionary are implemented as *linked lists* in a common dictionary buffer (a conventional memory array). A *list element* (which is available at a given physical address of the buffer) representing a node of a tree consists of the following *fields*:

- the signature (or character) associated with the node;
- the number of successors stored in the signature tree (limited to 3);
- a mask defining whether the successors are characters or signatures;
- a pointer addressing the successor nodes (address of the list element representing the first successor node; the other successor nodes are stored subsequently, after the first one); if there is no successor then a null pointer is assigned.

The character which is associated with the node is the physical address of the list element, in this way it has not be stored.

The placement of the linked lists in the dictionary buffer is performed by the preprocessor which builds the dictionary:

- The characters associated with the root nodes of the trees are exactly the signatures associated with these nodes (i.e., the physical address of a list element corresponding to a root node is the signature which is associated with this node). In this way, in the start phase of the compression algorithm, the signature tree is addressed directly by the signature.
- Since the nodes (which are not root ones) could be associated with arbitrary unique characters, the dictionary buffer is fully utilized (Figure 3):
  - list elements representing the root nodes are placed at the bottom of the buffer, at subsequent addresses (determined directly by the value of the signature);
  - list elements representing the successors of a given node are found at subsequent (neighboring) addresses, in this way a single pointer defines the set of successors.

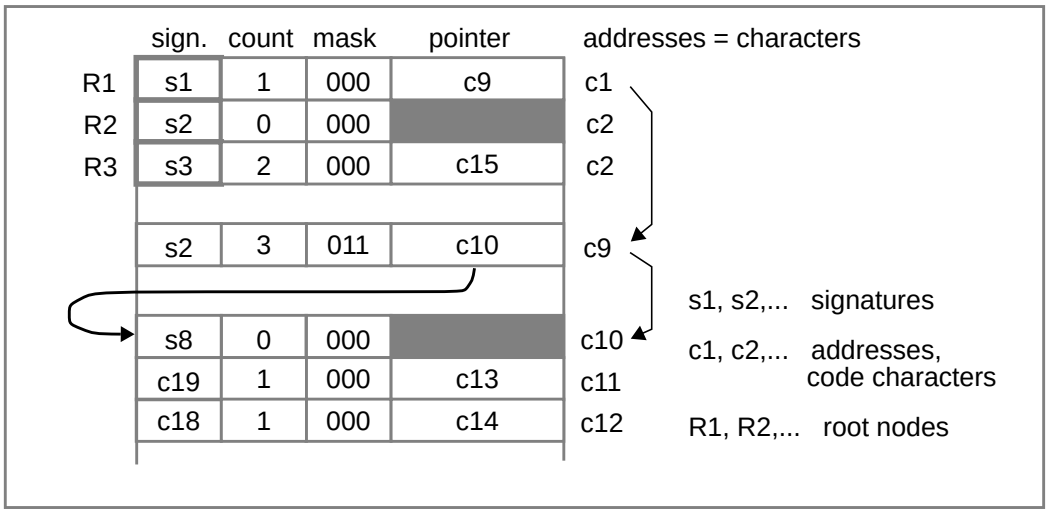

Figure 3 Implementation of the dictionary with embedded characters

2.5 Properties of the compression algorithm

The (enhanced) compression algorithm is real-time in the sense that the processing time of a signature is limited, independently whether the signature is included in a word or it is encoded separately. The transfer of signatures is not stopped or slowed down due to the difficulties or special cases of the compression. Theoretically, the number of levels of the embedding (characters in signature sequences encoded by other characters) is not limited. However, if a mismatch is detected by the algorithm then the storing of the character stack needs extra time proportional with the number of characters on the character stack. This is completely controlled by the construction of the dictionary, in this way the real-time properties are not violated.

The efficient hardware implementation is ensured by the following properties:
- No run-time building and modification of the dictionary is needed (it is predefined).
- The dictionary is stored in a form fully utilizing the dictionary buffer.
- The root of a signature tree is addressed directly by the signature.
- The successors of a node in a signature tree are addressed by a stored pointer, they are available at subsequent addresses.
- The examination of the possible successors requires a limited number of comparisons.

3 Compression of the signature sequence in a SEIS WP

The previous section presented a compression scheme using a predefined dictionary which can be easily derived analyzing the (high level) source text of the program to be executed. The dictionary should be downloaded into the compressor before the program run. In this way, starting new programs in the (multi-tasking) environment requires the downloading of new dictionaries which results in time and hardware overhead (storage of multiple dictionaries).

This section proposes a compression scheme which retains the simplicity of the previous scheme but universal in the sense that it does not need any predefined dictionary which is to be downloaded. The scheme is based on the SEIS assignment of signatures, thus it can be combined with signature checking by SEIS watchdog processors. In the following, first the SEIS signature assignment is described then the compression algorithm, its requirements and limitations are presented.

3.1 The SEIS signature assignment

In order to keep the evaluation of the run-time signatures simple, the SEIS signatures represent not only the statements of the program but also contain information about the valid (run-time) immediate successor signatures. Each SEIS signature (as a statement label) consists of 3 individual parts called *sublabels*. A signature is valid successor of a previous signature if and only if one of its sublabels is valid successor of one of the sublabels of the previous signature. The successor function of the sublabels is the natural function increasing the value of the sublabels by one.

A valid path in the CFG is represented by a sequence of signatures where each signature is a valid successor of the previous one. In this sequence, the subsequent signatures are connected

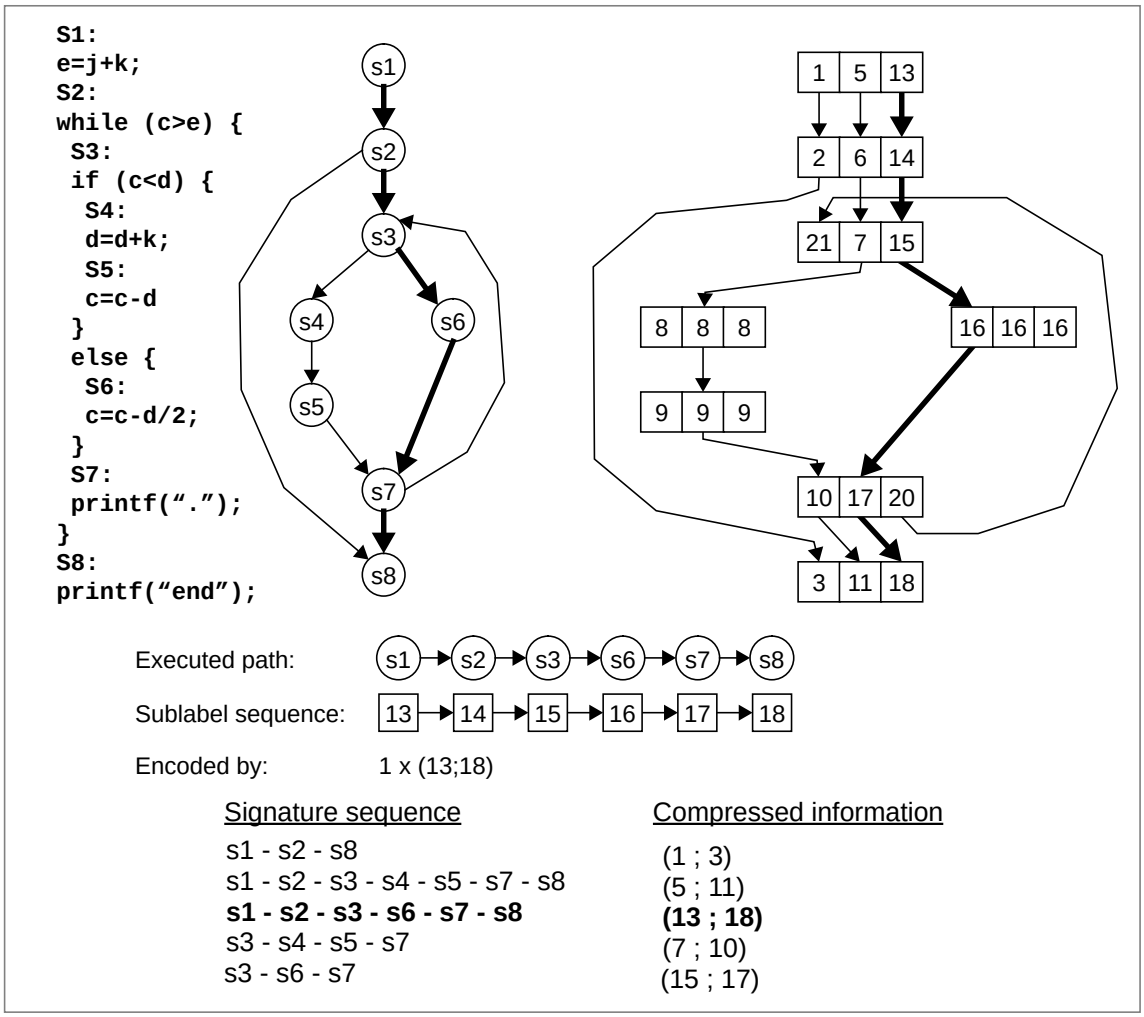

Figure 4 Assignment and compression of SEIS signatures

by successor sublabels (an edge of the CFG is associated with two unique sublabels, a startpoint sublabel and an endpoint sublabel in the signatures belonging to the connected nodes). Consider a signature in the run-time sequence. If the same sublabel connects the predecessor signature to the actual one and the actual signature to the successor one then the actual signature is called an *ordinary* signature in the sequence.

3.2 Compression algorithm

Each sublabel is unique in the signature set (within the limitations of the number of bits in the signature word), in this way one of the sublabels of a signature identifies the complete signature and thus a node of the CFG. Based on this fact, a run-time sequence of signatures can be easily compressed if all signatures in the sequence are ordinary ones. In this case, the sequence of signatures can be reduced to the sequence of the sublabels which connect the subsequent signatures. This sequence of sublabels is identified by the first and the last sublabel in the sequence (due to the deterministic successor function), in this way it can be encoded by these two values, independently of the number of sublabels in the sequence (Figure 4).

The compression algorithm examines whether the actual signature is an ordinary one. If it is ordinary then the sequence may continue, otherwise the actual sequence is reduced to a sublabel sequence which is encoded by its first and last sublabels. The compressed sequence (the pair of the two sublabels) is stored into the compression buffer.

1 ***Start phase***: The first signature of a sequence is stored in a temporary buffer. The next signature is received immediately. The sublabel of the first signature which connects it to this next one is stored as *start sublabel*, its successor in the next signature is marked as the *actual sublabel*. The following signature is processed in the *Encoding phase*.
If there is no sublabel that connects the first signature to the next one (e.g. this later one is an initial signature of a procedure called by the previous statement) then the first signature is stored (*Storing phase*, selecting an arbitrary sublabel of it) and the next one is processed in the *Start phase* as first signature of a new sequence.

2 ***Encoding phase***: As the actual signature is received, it is examined whether the previous signature is an ordinary one.
If the previous signature is connected to the actual signature by the actual sublabel then it is an ordinary signature. The successor of the actual sublabel becomes the new actual sublabel, the next signature is received and processed in the *Encoding phase*.
If the sublabel of the previous signature, which connects it to the actual signature, is not the actual sublabel then the sublabel sequence is terminated. The encoded sequence is stored into the compression buffer (*Storing phase*). The actual signature is processed in the *Start phase* as first signature of a new sequence.

3 ***Storing phase***: The compressed signature sequence is stored as the pair of the start sublabel and the actual sublabel.
If this pair is the same as the previous one stored in the buffer then only its counter has to be increased by one. Otherwise the actual pair has to be stored in the next element of the compression buffer (with *1* as counter value).

3.3 Properties and limitations of the SEIS compression

The construction of the SEIS CFG does not take into account the requirements of the compression as the edge sequences are defined mainly in the order of the syntactic occurrence and it is not guaranteed that ordinary signatures are assigned. However, the efficiency of the above defined SEIS compression can be further improved. Preferred paths of the program execution which are expected to be executed frequently (belonging to the words of the dictionary as defined in the previous Section) can be distinguished by assigning subsequent ordinary signatures to the nodes (*path optimization*). To do this, transformations have to be executed on the CFG before the assignment of the sublabel values, still preserving the structural properties of the CFG (i.e. not introducing additional paths). The following steps can be defined (Figure 5):

- Shuffling the output edges of a node, i.e. transposing the startpoint sublabels in the corresponding signature.
- Shuffling the input edges of a node, i.e. transposing the endpoint sublabels in the corresponding signature.
- Introducing duplicated edges between nodes of the CFG.

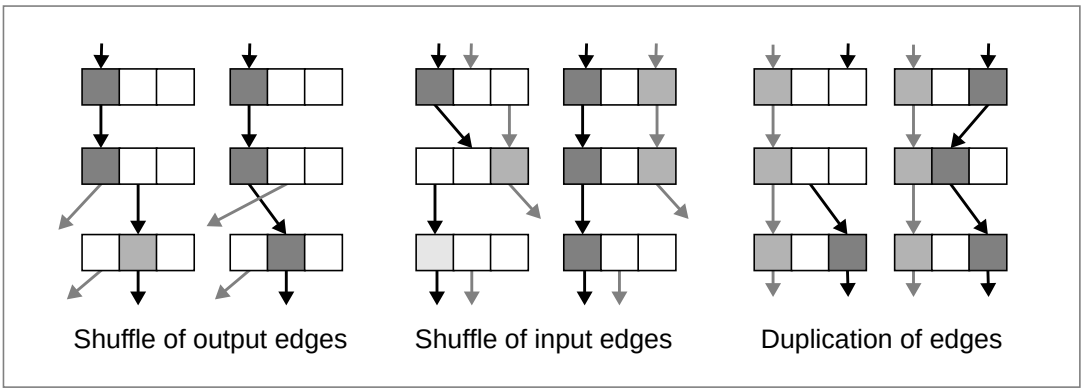

Figure 5 Path optimization in the SEIS CFG

The first two transformations produce ordinary signatures in a given path. The third transformation (which can be followed by the first two ones) enables a signature to be embedded in multiple different signature sequences. In the actual implementation of SEIS, the following constraints have to be taken into account during the path optimization:

- The number of sublabels of a signature is limited to 3, thus the number of input/output edges of a node is limited as well. (This limit of sublabels is proved to be satisfactory for programs in common structural languages like C, Pascal, Modula-2). Consequently, a signature can be embedded in maximum 3 different compressible run-time sequences.
- The number of input/output edges of nodes belonging to special statements (exception cases in the structural languages, like *goto, break* etc.) is further limited. Additionally, in these nodes the order of the edges (how the output edges follow the input edges) is constrained. Due to these constraints, in most of the cases the necessary transformations can not be executed thus the signatures belonging to these nodes terminate the signature sequences.
  The special statements and the constraints of the input/output edges are analyzed in details in [4].

Due to the limitations of the path optimization in the SEIS CFG, the optimal path selection and encoding can not be performed in all cases. The drawback is especially significant if there are more than 3 execution paths (of about the same probability) in the body of a frequently executed iteration. In these cases the general compression algorithm provides better results since there are no limitations in the path selection and encoding. However, the lack of dictionary and information downloading makes the SEIS compression still attractive.

4 Preliminary measurement results

The real-time signature compressor is intended to be built using an FPGA circuit (Xilinx 3000 series) which needs only an interface to receive signatures and a memory array to store the compression buffer (and the dictionary in the general case). Since the FPGA is programmable in run-time, both structures can be downloaded and evaluated. The fast compression algorithm and the low hardware overhead enable the circuit to be built into the conventional watchdog-processor unit [5]. The preliminary measurements were performed using software simulation.

4.1 Compression of general signatures

The effectiveness of the compression depends heavily on the optimal selection of the words, i.e. on the construction of the dictionary. To highlight this effect, the compression rate was measured building dictionaries of different size. The benchmark program was a multigrid based solver of differential equations, with reduced number of signatures (in average, every 5th statement was associated with a signature). First the paths inside of the iteration loops were encoded then additional paths as well. The results are given in Table 1. The iteration loops of the solver are relatively small, thus the compression rate is sensitive for small changes in the dictionary (as new paths are entered). The run of the iterations is data dependent since for different input parameters (number of levels), the same changes in the dictionary result in different effects.

Benchmark	Without compression	Dictionary size			
		85	89	95	116
multigrid 3	1,715 100%	1,181 69%	776 45%	692 40%	607 35%
multigrid 5	32,391 100%	15,914 49%	4,622 14%	4,118 13%	3,981 12%

Table 1 Size of the compressed trace vs. dictionary size

4.2 Compression of SEIS signatures

The effectiveness of the SEIS compression depends on the structure of the CFG, i.e. on the path optimization. The following measurements were made using various benchmark programs without additional path optimization (only the original SEIS encoding algorithm was executed which encodes the paths looking for maximum loops in the CFG in the order of syntactic occurrence). The results are satisfactory even in this case (Table 2). Signature sequences belonging to simple iterations and long statement sequences are compressed effectively (the compression becomes better increasing the number of steps in the iteration of the multigrid benchmark). Nested loops and complex control structures make the compression difficult.

Benchmark	Number of run-time signatures	Size of the compressed trace	Compression rate
multigrid 3	3,993	932	23%
multigrid 5	79,005	12,884	16%
multigrid 7	1,254,821	157,936	13%
whetstone	119,633	38,893	33%
dhrystone 100	12,288	3,803	31%
linpack	11,825,895	728,441	6%

Table 2 SEIS compression results

5 Support of diagnosis

The compression buffer stores a limited number of signatures in a compacted form. If an error is detected, the execution of the program is stopped and the compression buffer can be accessed by the checked computer or by external devices as part of the diagnostic procedure. The signature sequence preceding the error is available, in this way the sequence of statements executed before the error can be derived and analyzed.

The following considerations can help the successful diagnosis:

- If the error is reproducible then the dictionary can be redefined on the basis of the contents of the compression buffer (new paths can be encoded which were not included in the dictionary), in this way a longer signature sequence can be stored.
- If the program reaches a well-defined stable point (e.g. after a commitment; after a checkpoint generation; if the initial state is reached again) then the compression buffer can be reset and the compression is restarted. In this way the compression buffer contains exactly the signature sequence after the stable point in the execution.
- If a selected set of the input events of the checked program (e.g. interrupts, communication with other processes, input from peripherals, time events) is associated with signatures then input-specific or real-time constraints can be diagnosed as well.

The statements executed before the error are presented in a graphical environment similar to the one of the common debuggers: the statements or statement sets of the program execution are highlighted in the source text simulating an automatic trace or a single step execution controlled by the user.

The above mentioned environment can help the input-domain based test of programs as well. Since the signatures identify the possible paths of the program execution, it can be investigated whether a given test set covers all of the possible branches of the program. The signatures *not transferred* to the WP during the test identify the branches/paths which were not executed.

References

[1] Mahmood, A.; McCluskey, E. J.: Concurrent Error Detection Using Watchdog Processors - A Survey. IEEE Transactions on Computers 37, 160-174 (1988)

[2] Lu, D. J.: Watchdog Processors and Structural Integrity Checking. IEEE Trans. on Comp. 31, 681-685 (1982)

[3] Pataricza, A.; Majzik, I.; Hohl, W.; Hönig, J.: Watchdog Processors in Parallel Systems. Microprocessing and Microprogramming Vol. 39 (Proc. Euromicro'93, 19th Symposium on Microprocessing and Microprogramming, Barcelona, 1993), pp 69-74, 1993

[4] Majzik, I.: SEIS: A Program Control-Flow Graph Encoding Algorithm for Control Flow Checking. Technical Report No. TUB-TR-94-EE14, Technical University Budapest, Hungary, 66 pages, 1994

[5] Majzik, I.; Pataricza, A.; Dal Cin, M.; Hohl, W.; Hönig, J.; Sieh, V.: Hierarchical Checking of Multiprocessors using Watchdog Processors. Springer LNCS 852, Springer Verlag, Berlin Heidelberg, pp 386-403, 1994