PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 40, NO. 2, PP. 87-103 (1996)

SOFTWARE DIAGNOSIS USING COMPRESSED SIGNATURE SEQUENCES

István MAJZIK

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

Received: June 14, 1996

Abstract

Software monitoring and debugging can be efficiently supported by one of the concurrent error detection methods, the application of watchdog processors. A watchdog processor, as a co-processor, receives and evaluates signatures assigned to the states of the program execution. After the checking, it 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 efficiently utilized if the signature sequence is compressed. In the paper, two real-time compression methods are presented and compared. The first one uses predefined dictionaries, while the other one utilizes the structural information encoded in the signatures.

Keywords: software diagnosis, compression, monitoring, watchdog processors.

1. Introduction

Post-mortem diagnosis of embedded (real-time) application programs is often difficult, since in most of the cases the diagnosis is supported only by some static information like a memory dump. The system designer is interested in the trace of the erroneous program, i.e. in the sequence of statements executed by the program before the error.

Complex and costly monitoring systems are implemented to collect and store the trace data, often modifying the original operating environment and interfering with the timing of the monitored programs. The majority of the commonly used debugging tools requires architecture specific hardware. For example, the RED method [1] uses a dedicated co-processor for collecting trace data, the DCT toolset [2] utilizes a hardware bus monitor. By the software approach, the source of the monitored program is modified, inserting extra instructions which collect data. In ART [3] a special reporting process is needed (permanently integrated into the application software), in [4] or [5] the source is modified before compilation. Hy-

brid techniques like [6] and [7] add a few instructions to the code of the program but the information is collected by dedicated hardware.

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 easily be extended to support the trace based 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 efficiently be 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 application of *watchdog processors*. A watchdog processor (WP [8]) is a relatively simple coprocessor monitoring the state of the system using signatures, compact abstractions of the system state. In the assigned signatures methods [9], the checked program is modified at compilation time by a preprocessor in 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 being not a valid successor of the previous signature is found then a control-flow error is detected: the program entered an erroneous state.

The signatures assigned by the preprocessor identify uniquely the states of the program. In the default case, each individual statement of the program is associated with a unique signature, but additional reduction phases can merge branch-free statement sequences into a block labelled one 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 restored later. 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 this sequence is too long to be stored in full extent for practical programs. 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 effi-

ciency 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 subsequences, the compression is possible.

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 diagnostic purposes.

In Section 2 the compression of a general signature sequence is examined. 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 [10]) method allows the definition of a general compression scheme. In this case no dictionary is needed, the run-time signature sequence can be compressed without downloading any program-specific dictionary. The compression technique 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 successive 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 and sophisticated hardware. We propose a method which allows a simple 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 utilizes 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 dependences of the program run. But, since the control flow graph of the program is known, some program paths, which are assumed to be executed frequently, can be identified. This way, signature sequences originating from the execution of these program paths can define the words of the dictionary. The execution of the following structures can be taken into account:

- iterations (loops);
- long (branch-free) sequences of instructions;
- normal branches of selections (exception should rarely occur);
- frequently called small procedures.

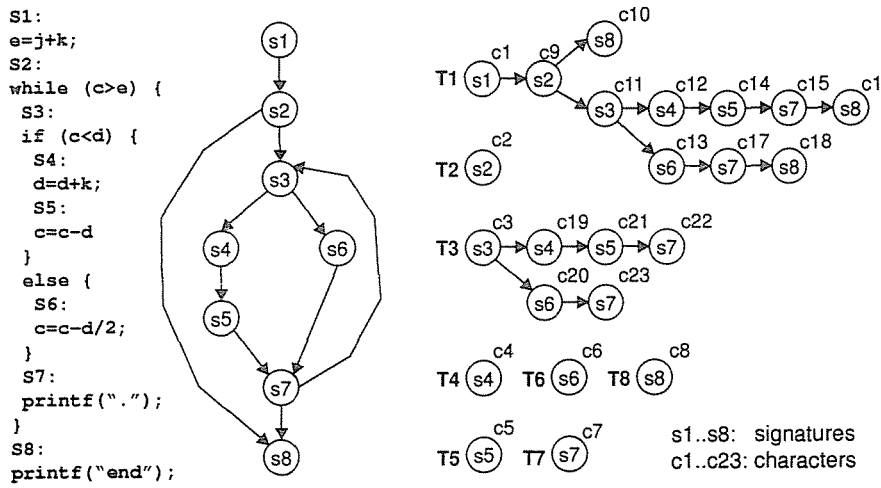

Fig. 1. A program CFG and the corresponding signature trees

The preprocessor which analyzes the program text can define the dictionary, identifying the above structures and deriving the signature sequences associated with them.

To compress a run-time signature sequence, an extension of the traditional WP, a *signature compressor* is built. It receives the signatures, performs on-line compression and stores the compacted sequence in the WP-internal compression buffer. In the following, first the structure of the predefined dictionary, then the compression algorithm and its properties are presented.

2.1 The Dictionary

The words (i.e. signature sequences) in the dictionary are associated with frequently executed program paths, but (due to the limited resources) regularly do not cover all of the possible paths of the program execution. The signature sequences belonging to paths not mapped directly to words are to be stored in the compression buffer as well. To ensure this,

- 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 in the sequences according to the CFG, and so on until the endpoints of the tree. In this way a node of the tree identifies a unique signature sequence starting with the root signature and ending at the given node.

The nodes of the trees – and, consequently, the signature sequences in the signature trees – 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 (*Fig. 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 *s3-s4-s5-s7* is a path in the tree starting with *s3*, and *s4* always follows *s3*, then the postfix *s4-s5-s7* is not represented by a path in the tree starting with *s4*. Accordingly, most of the trees consist of only the root signature. The nodes, which are included in several words, are represented by nodes in several signature trees. In *Fig. 1* an example

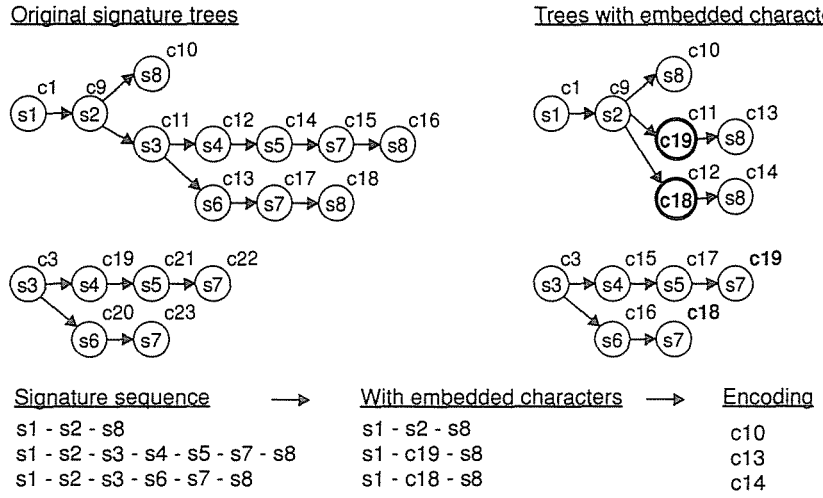

Fig. 2. Signature trees with embedded characters

CFG and the corresponding signature trees are presented. The possible paths of the iteration are encoded separately since they are expected to be executed frequently. Two complete paths are also covered by words.

There are (longer) paths in the CFG which share common subpaths. Accordingly, the signature sequences corresponding to these subpaths are included in various longer words. To reduce the size of the dictionary, the 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 the characters should be used as embedded characters which represent a path (signature sequence) *from the root to the end* (leaf) of a signature tree. The signature trees depicted in *Fig. 2* illustrate how the size of the dictionary is reduced using embedded characters.

2.2 The Compression Algorithm

The task of the compression algorithm is to find the longest word which is encoded by a single character. (Note that 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 corresponding character is stored in the compression buffer. The compres-

sion buffer is a linear array of elements, each element consists of two fields: one storing the character and a second one counting the subsequent occurrences of the character.

The compression begins in the *Start phase* then continues in the *Encoding phase*: signatures are received and processed looking for the character which encodes the sequence. Found a character that should be stored in the compression buffer then the *Storing phase* is called. A *character stack* is maintained by the algorithm (storing the predecessors of 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 root of this tree is the actual node. The next signature is received and processed in the *Encoding phase.*

2 **Encoding phase**: The successors of the actual node are accessed.
   - 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 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 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 completed, the actual character is stored in the compression buffer (*Storing phase*) and the actual signature is processed in 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 in 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 completed. 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 a *separate word.* The actual character is stored in the compression buffer (*Storing phase*), the actual signature is processed in 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 this actual node are addressed and compared with the actual character:

- If there is a character successor of the actual node which equals 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 removed, and the actual signature is processed in the *Encoding phase* (continuing the encoding of the word).
- If there is no character successor of the actual node which equals the actual character then the actual word is not the embedded word: the original word is completed 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 in the *Start phase* (as a first signature of a new word).

3 **Storing phase**: If a character is found that encodes a 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 is increased by one. Otherwise the actual character is stored in the next element of the compression buffer (using 1 as the initial counter value).

2.3 Implementation of the Dictionary

For the sake of effectiveness and high speed of the signature compressor, 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 address of the buffer) representing a node of a tree consists of the following fields:

- the signature (or embedded character) associated with the node;
- the number of successors stored in the signature tree (limited to 3; note that the structural properties of the control-flow graphs of programs enable this limitation as most of the signatures have only 1 or 2 valid immediate successors);
- 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; the other successors are stored in

succession after the first one); if there is no successor then a null pointer is assigned.

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

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

- The characters associated with the root nodes of the trees are exactly the signatures associated with these nodes (i.e. the address of a list element corresponding to a root node is the signature which is associated with this node). This way, in the start phase of the compression algorithm, the signature tree is accessed directly by the signature itself using it as the address.
- Since the other nodes (which are not root ones) can be associated with arbitrary (but unique) characters, the dictionary buffer is fully utilized (*Fig. 3*):
  - list elements representing the root nodes are placed at the bottom of the buffer, at the addresses determined directly by the value of the signature;
  - list elements representing the successors of a given node are found at successive (neighbouring) addresses, in this way the set of successors is given by a single pointer.

2.4 Properties of the Compression Algorithm

The compression algorithm is real-time in the sense that the processing time of a signature is bounded, 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 need of compression.

Theoretically, the number of levels of the character embedding is not limited. However, if a mismatch is detected by the algorithm then storing the character stack needs extra time proportional with the number of characters on the stack. It is controlled completely by the construction of the dictionary, in this way the requirements of the real-time execution can be taken into account.

The efficient hardware implementation is ensured by the following design aspects:

- No run-time construction 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.
- 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 easily be 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 requires the downloading of the new dictionaries which results in time and hardware overhead (storage of multiple dictionaries), especially in multi-tasking environments.

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

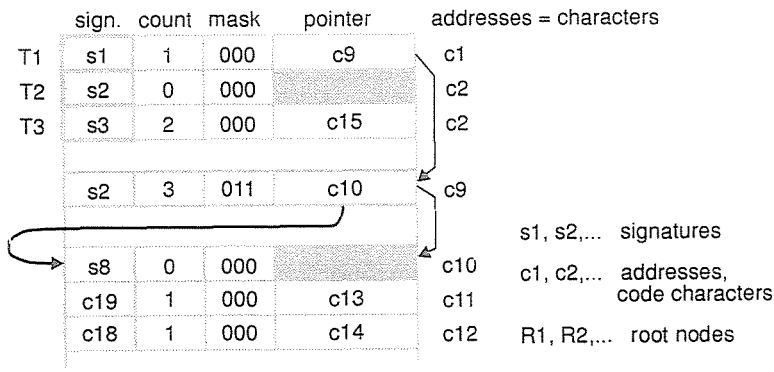

Fig. 3. Implementation of the dictionary

3.1 The SEIS Signature Assignment

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 a valid successor of a previous signature if and only if one of its sublabels is a valid successor of one of the sublabels of the previous signature. The successor function of the sublabels is the 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 successive signatures are connected 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 a *compressible* signature in the sequence.

3.2 The 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 compressible ones. In this case, the sequence of signatures can be reduced to the sequence of sublabels which connect the successive signatures. This *sequence of sublabels* is identified by the *first and the last sublabels* in the sequence (due to the deterministic successor function), in this way it can be encoded by these two values, independently from the number of sublabels in the sequence (*Fig. 4*).

The compression algorithm examines the run-time signatures. If the actual signature is a compressible one then the sequence may continue, otherwise a subsequence is found which is reduced to a sublabel sequence encoded by its first and last sublabels. The compressed sequence (the pair of the two sublabels) is stored in the compression buffer.

1 **Start phase**: The first signature of a sequence is stored in a temporary buffer. The next signature is received. The sublabel of the first signature which connects it to this next one is stored as *start subla-*

bel, 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) 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 a compressible one.
If the previous signature is connected to the actual one by the actual sublabel then it is a compressible signature. The successor of the actual sublabel becomes the new actual sublabel, the next signature is received and processed in the *Encoding phase*.

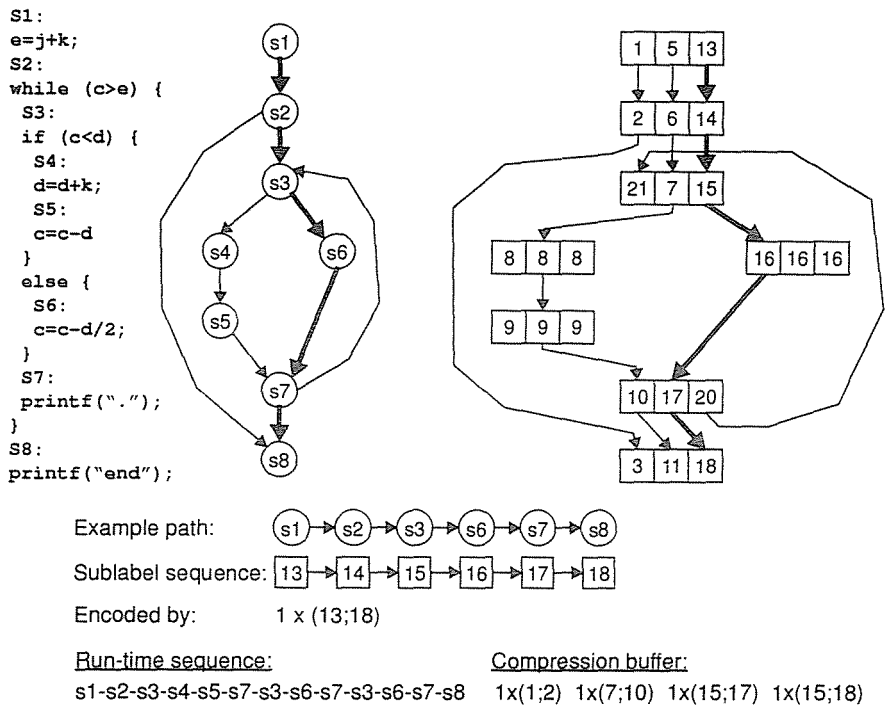

Fig. 4. Assignment and compression of SEIS signatures

If the sublabel of the previous signature, which connects it to the actual signature, is not the actual one 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 compression buffer then only its counter is increased by one, otherwise the actual pair is stored in the next element of the buffer (with 1 as initial counter value).

3.3 Properties and Limitations of the SEIS Compression

In its original form, the assignment of the SEIS signatures did not take into account the requirements of the compression. The edge sequences were defined mainly in the order of the syntactic occurrence, the algorithm was not optimized to assign compressible signatures. The efficiency of the above defined SEIS compression can be further improved by *path optimization:* preferred paths of the program execution, which are expected to be executed frequently, can be distinguished by assigning compressible signatures to the nodes. To do this, transformations are executed on the CFG before the assignment of the sublabel values, still preserving the structural properties of it (i.e. not introducing additional paths). The following transformations are defined (*Fig. 5*):

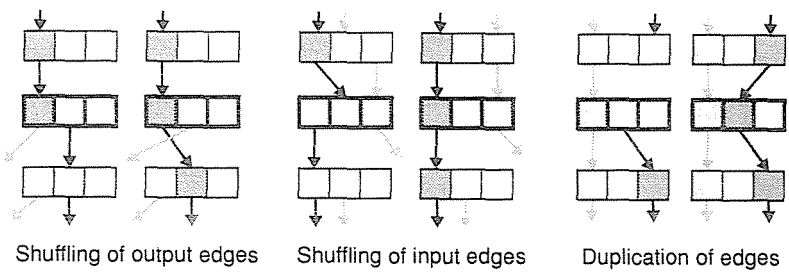

Fig. 5. Path optimization in the SEIS CFG

- Shuffling the input or output edges of a node, i.e. reversing the sublabels in the signature;
- Inserting duplicated edges between adjacent nodes of the CFG.

The first two transformations produce compressible signatures in a given path, the third transformation (which can be followed by the first

two ones) enables a signature to be included in several 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, thus the number of input/output edges of a node is limited. (This limitation of sublabels is proved to be still 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 (exceptional cases in the structural languages, like *goto, break*, etc.) is further limited. Additionally, in these nodes the way how the output edges follow the input edges is fixed [11]. Due to these constraints, in most of the cases the necessary transformations cannot be executed, thus the signatures belonging to these nodes terminate the sequences of compressible signatures.

Due to the limitations of the path optimization in the SEIS CFG, the optimal path selection and encoding cannot 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 dictionaries makes the SEIS compression still attractive.

4. Measurement Results

The real-time signature compressor can be built using a single FPGA circuit (Xilinx 3000 series) which needs only an interface to receive signatures and an external memory array to store the compression buffer (and the dictionary in the general case) [12]. The fast compression algorithm and the low hardware overhead enable the circuit to be built into a conventional watchdog processor unit [13]. The preliminary measurements were performed using software simulation.

4.1 Compression Using a Predefined Dictionary

The effectiveness of the compression depends on the optimal selection of the words, i.e. on the construction of the dictionary. To highlight this effect, the compression rate was measured constructing dictionaries of different size. The benchmark program was a multi-grid based solver of differ-

ential equations, with reduced number of signatures (in average, every 5th statement was associated with a signature). First the signature sequences belonging to paths inside the iteration loops were encoded then additional paths as well. The results (number of occupied elements in the compression buffer) are given in *Table 1.* The iteration loops of the solver are relatively small, thus the compression rate is sensitive to small changes in the dictionary. The iterations are data dependent, since for different input parameters (number of levels) the same changes in the dictionary result in slightly different effects.

Table 1
Size of the compressed trace vs. dictionary size

Benchmark	Without compression	Dictionary size			
		85	89	95	116
multigrid 3	1.715	1.181	776	692	607
	100%	69%	45%	40%	35%
multigrid 5	32.391	15.914	4.622	4.118	3.981
	100%	49%	14%	13%	12%

4.2 Compression of SEIS Signatures

The effectiveness of the compression depends on the structure of the CFG and on the result of the path optimization. The following measurements were performed without additional path optimization (the original SEIS encoding algorithm was executed which encodes the paths looking for loops in the CFG in the order of syntactic occurrence). The results are satisfactory even in this case (*Table 2*). Signature sequences belonging to iteration loops and long statement sequences are compressed efficiently (increasing the number of steps in the iteration of the multi-grid benchmark, the compression becomes better). Nested loops and complex control structures make the compression difficult.

5. Support of Diagnosis

The compression buffer stores a limited number of signatures in a compacted form. If an error (or other trigger event) is detected then the execution of the program is stopped and the compression buffer can be accessed either by the checked computer itself or by external devices. On the basis

Table 2
SEIS compression results

Benchmark	Number of run-time signatures	Size of the compressed trace	Compression rate
multigrid 3	3.993	1008	25%
multigrid 5	79.005	13.386	17%
multigrid 7	1.254.821	160.073	13%
whetstone	118.793	38.895	33%
dhrystone 100	12.288	3.705	30%
linpack	11.825.895	603.300	5%

of the stored signatures the sequence of statements executed before the error can be derived and analyzed (as part of the diagnostic procedure).

The successful diagnosis can be supported by the following extensions:

- 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. commitment, checkpoint generation, the initial state) then the compression can be restarted. In this way the compression buffer contains the signature sequence received after the stable point in the execution.
- If a selected set of input events of the checked program (e.g. interrupts, communication with other processes, input from peripherals, timing 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 an environment similar to the one of common debuggers: the statements or statement sets of the program execution are highlighted in the source text simulating automatic trace or single step execution.

The above mentioned environment can help the input-domain based testing 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.

6. Conclusion

In our paper a new approach of signature-based monitoring and debugging of programs is proposed. It is shown that one of the concurrent error de-

tection techniques, the application of watchdog processors can be extended easily to support trace based diagnosis. Signatures which identify the states of the program can be stored efficiently in a trace buffer, in a compressed form. Two approaches for the real-time compression of the run-time signature sequence are presented and evaluated. Our future work is concentrated on the refinement of the diagnostic environment and on the improvement of the path optimization in the case of SEIS signature assignment.

References

1. HILL, C. R.: A Real-Time Microprocessor Debugging Technique. In *ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on High Level Debugging*, pp. 145–148, 1983.
2. BHATT, D. – GHONAMI, A. – RAMANUJAN, R.: An Instrumented Testbed for Real-Time Distributed Systems Development. In: *IEEE Symp. on Real-Time Systems*, pp. 241–250, 1987.
3. TOKUDA, H. – KOTERA, M. – MERCER, C. W.: A Real-Time Monitor for a Distributed Real-Time Operating System. *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 68–71, 1988.
4. TAI, K. C. – CARVER, R. H. – OBAID, E. E.: Debugging Concurrent ADA Programs by Deterministic Execution. *IEEE Trans. on Software Engineering*, Vol. 17. No. 1, pp. 45–63, 1991.
5. MAEHLE, E. – OBELOER, W.: Delta-T: A User-Transparent Software Monitoring Tool for Multi-Transputer Systems. *Microprocessing and Microprogramming*, Vol. 35, pp. 245–252, 1992.
6. CALVEZ, J. P. – PASQUIER, O.: Real-time Behavior Monitoring for Multi-Processor Systems. *Microprocessing and Microprogramming*, Vol. 38, pp. 213–220, 1993.
7. HABAN, D. – WYBRANIETZ, D.: A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Trans. on Software Eng.*, Vol. 16/2, pp. 197–211, 1990.
8. MAHMOOD, A. – MCCLUSKEY, E. J.: Concurrent Error Detection Using Watchdog Processors – A Survey. *IEEE Transactions on Computers*, Vol. 37, pp. 160–174, 1988.
9. LU, D. J.: Watchdog Processors and Structural Integrity Checking. *IEEE Trans. on Comp.* Vol. 31, pp. 681–685, 1982.
10. 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.
11. 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.
12. LÖVÉSZ, L.: Signature Based Software Diagnosis. Project laboratory report, Technical University of Budapest, Dept. of Measurement and Instrument Engineering, (in Hungarian), 1994.
13. 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.