Software Monitoring and Debugging Using Compressed Signature Sequences

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

Abstract

Signature based error detection techniques (e.g. the application of watchdog processors) can be easily extended to support software debugging. The run-time sequence of signatures is stored in an extension of the traditional checker. As the signatures identify the states of the program, a trace of the statements executed by the checked processor 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 general method uses predefined dictionaries, while the other one utilizes the structural information encoded in the signatures.

1. Introduction

Debugging or post-mortem diagnosis of complex, embedded (real-time) application programs is difficult, since it is supported typically only by 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 the program executed before the error.

The complex monitoring systems intended to collect the trace data may even modify the original operating environment and timing of the monitored programs. The majority of monitoring and debugging tools (e.g. the RED [4] or DCT [1] methods) require dedicated hardware. By software approaches, the source of the monitored program is modified, inserting extra instructions which collect trace data [6] or extra processes [12]. Hybrid techniques like [2] or [3] combine software modification and hardware-based information collection.

Some error detection techniques widely used 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 extended easily to support trace based diagnosis and debugging of programs.

Dependable applications require continuous, concurrent run-time error detection mechanisms to highlight transient errors causing disturbances in data and control flow. Errors in data can be efficiently detected (and even corrected) by redundant encoding, while the most efficient method for the detection of control-flow errors is the application of watchdog processors (WPs, [7]). WPs are relatively simple coprocessors monitoring the state of the system using signatures, compact abstractions of the system state. In the assigned signatures methods [5], the checked program is modified at compilation time in such a way, that during the run the signatures are transferred to the WP. (A 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 reference control flow graph (CFG) extracted by the preprocessor. If the WP finds a signature which is not a valid successor of the previous one then a control-flow error is detected.

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 labeled by a single, joint signature. In this way, the run-time sequence of signatures contains the information necessary to reconstruct the execution of the program, the trace of statements. However, the original error detection mechanism does not store this sequence of run-time signatures in the WP.

If the run-time signature sequence is stored in the *diagnostic extension* of a traditional 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. A trade-off between the efficiency and moderate cost is to implement a logic analyzer-like circular buffer storing a limited log of signatures. The utilization of the buffer can be further improved by some kind of information compression of the signature sequence before storing the log. Since the majority of signatures originates from repetitive signature sub-sequences (corresponding to iteration loops, frequently called procedures), the efficient compression is possible.

The basic idea is summarized as follows: Signatures are assigned to the states of the application programs by a preprocessor. The diagnostic WP receives and compresses the run-time sequence of signatures, the compressed sequence is stored in a circular buffer. If a trigger condition (e.g. an error) is detected then the registration is stopped and the buffer can be read by the diagnosis program or an external supervisory computer. 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 detection is available for diagnosis and debugging purposes. The advantage of our solution is, that even complex (multi-tasking) applications, time and data dependent control flow (scheduling, interrupt handling) can be analyzed, using the very same hardware as for on-line error detection.

Our paper concentrates on the signature compression techniques. We propose two schemes allowing an extremely simple real-time hardware compressor unit, utilizing the a priori knowledge of the structure of the program to be analyzed. The first scheme uses a predefined dictionary, which is constructed on the basis of the control flow graph of the program to be monitored (Section 2). The second one utilizes the redundant structure of the signatures assigned by the SEIS (*Signature Encoded Instruction Stream*, [11]) method. In this case no dictionary is required, the run-time signature sequence can be compressed without downloading any program-specific dictionary. This interesting (suboptimal) compression is discussed in Section 3. At the end, measurement results (Section 4) and support of diagnosis (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 that of universal encoding. In our case, the message is the run-time sequence of signatures, the message alphabet consists of the valid signatures while the encoding alphabet contains a fixed number of *characters*. The signature sequence is divided into *words* (slices of the signature sequence) of varying length 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 algorithms) the dictionary is built concurrently with the transfer of the message. The run-time construction of the dictionary is time-consuming and needs a fast, difficult and sophisticated hardware.

In our case, the dictionary is constructed during preprocessing, when the control flow graph is extracted and the signatures are assigned to the states of the program. Before the start of the program, the dictionary is downloaded into the diagnostic WP. The compression mechanism uses 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). The special structure of the dictionary ensures the simplicity of the compression hardware and the real-time processing of signatures. The efficiency of the compression depends on the definition of the dictionary, i.e. on the optimal selection of the words.

The actual signature sequence is unpredictable in the preprocessing phase due to the data dependency of the program run. But, since the control flow graph of the program is already known, program paths assumed to be executed frequently can be identified by the preprocessor. In this way, signature sequences originating from the execution of these program paths define the words of the dictionary. The following structures can be taken into account:

- bodies of iterations (loops);
- 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.

The diagnostic extension of the WP consists of the signature compressor, the dictionary buffer and the signature buffer. In the following, first the structure of the dictionary is described, then the compression algorithm and its properties are presented.

2.1. Structure of 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. There are signatures which do not belong to paths mapped directly to words. To store them in the buffer,

- each valid (single) signature is encoded by a unique character;
- subpaths of encoded paths, i.e. prefixes of signature sequences mapped to words are also encoded by unique characters.

The dictionary consists of characters which encode words (signature sequences) corresponding to selected program execution paths. Words starting with a given signature (i.e. some subpaths of the program execution starting in a given state) are represented by a *signature tree*. The given initial signature is the root of the tree, followed by its immediate successors in the CFG involved in (one of) the words. A node of the signature tree identifies a signature sequence starting with the root signature and ending at the given node. Words defined by the preprocessor are represented by a set of signature trees with unique root signatures. Each valid signature is a root of an individual tree (if a signature is not an initial one in any word then it forms a tree consisting of this signature (as a root) only).

The nodes of 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 (Figure 1). The above mentioned requirements are satisfied: each signature, as root of a tree, is encoded by a unique character, and prefixes of words are also encoded by characters.

Postfix of a given 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 signatures, which are included in several words, are represented by nodes in several signature trees. In Figure 1 an example CFG and the corresponding signature trees are presented. Subpaths in the iteration are encoded separately since they are expected to be executed frequently. Two complete paths are also covered by words.

For the sake of effectiveness and high speed of the compressor hardware, the signature trees of the dictionary are implemented as *linked lists* in the dictionary buffer. A list element representing a node of a tree consists of the following fields:

- the signature associated with the node;
- the number of successors stored in the signature tree (limited to 3; note that the structural properties of the CFGs of programs enable this limitation as most of the signatures have only 1 or 2 valid immediate successors);
- a pointer addressing the successor nodes (address of the list element representing the first successor node, the other successor nodes are stored successively, 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 *address* of the list element in the buffer, in this way it has not to be stored.

The dictionary as a linked list is constructed by the preprocessor taking into account the efficient utilization of the buffer. The characters encoding the root nodes of the trees are identical to the signatures associated with these nodes. This way a signature tree is easily accessed since the address of the list element corresponding to the root node is exactly the first (root) signature of the tree. These nodes are placed at the bottom of the buffer, at successive addresses (in our case the signatures are increasing numbers, in the order of syntactic occurrence in the program text [10]). The other nodes (which are not root ones) are associated with arbitrary but unique characters, in such a way, that the successors of a given node are found at successive addresses (this way a single pointer defines the set of successors).

2.2. The compression algorithm

The task of the compression algorithm is to find the longest word which is encoded by a single character. (As a default worst case, each signature is encoded by a separate character.) If the longest word is found then the encoding character is stored in the compression buffer. The compression buffer is a linear array of elements consisting of two fields: one storing the character and a second one storing the subsequent occurrences of the character.

The compression algorithm begins 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 characters encoding words are 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 accessed. 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 (i.e. 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 completed. The character associated with the actual node is stored into the compression buffer (*Storing phase*), the actual signature is processed in the *Start phase*.

Figure 1. An example program, its CFG and the corresponding signature trees

3. *Storing phase*: If a word is completed then the character encoding this word, i.e. 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 is increased by one, otherwise the actual character is stored in the next element of the compression buffer (with 1 as initial counter value).

   The actual signature is processed as first signature of the next word (*Start phase*).

The compression algorithm can be enhanced. There are (longer) paths in the CFG which share common subpaths. Accordingly, the signature sequences corresponding to these subpaths are 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. For example, let consider the signature tree $T1$ in Figure 1. The subpaths $s3 - s4 - s5 - s7$ and $s3 - s6 - s7$ of $T1$ can be replaced by $c22$ and $c23$, respectively, since they are encoded in $T3$. (Note that in iterative programs it is often required to encode long paths which contain sub-paths of embedded cycles.)

Using the enhanced compression algorithm, the structure of the dictionary is modified: a mask is introduced which distinguishes the signature or character successors of a given node (root nodes always represent signatures). The algorithm also maintains an additional *character stack* storing the predecessors of embedded characters. The number of levels of the character embedding is limited when the dictionary is constructed.

The compression algorithm is real-time in the sense that the time needed to process a signature is bounded, independently whether the signature is included in a word or it is encoded separately. The fast and efficient hardware implementation is ensured by the following design aspects:

- No run-time construction or modification of the dictionary is needed. It is stored in a form fully utilizing the dictionary buffer.

- The search and compare operations in the dictionary can be processed fast, since the signature trees are accessed directly by the root signatures and the successors of a given signature are addressed by a stored pointer. The examination of the possible successors requires a bounded number of comparisons.

3. Compression of the SEIS signature sequence

The previous section presented a compression scheme using a predefined dictionary which can 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 new dictionaries which results in time and hardware overhead, especially in multitasking environments. The second compression algorithm retains the simplicity of the previous scheme and additionally eliminates the use of a predefined dictionary. The scheme is based on the SEIS assignment of signatures [8], 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 discussed.

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 their valid immediate successors. Statements of a branch-free program path are encoded by a series of successive incremental numbers, in branches the alternative paths require starting of additional, separated series. A signature assigned to a statement involved in more than one path consists of more individual parts called *sublabels*. The successor signatures are connected by successor sublabels similarly like the tiles in the game of domino. The SEIS encoding algorithm ensures that each signature consists of a limited number of sublabels. The sublabels are unique in the encoding, in this way a given sublabel identifies the signature and thus the statement of a program. If a slice of the run-time sequence of signatures represents a program path encoded by a series of successive numbers, it can be compressed by storing only the first and last sublabels of the series, in a similar way as an interval is defined by its endpoints.

3.2. The compression algorithm

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. 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.

Each unique sublabel identifies the complete signature and thus the corresponding state (statement) of the program. 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 the sublabels which connect the successive 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 2).

The compression algorithm examines whether the actual signature is a compressible one. If it is compressible then the sequence may continue, otherwise the actual sequence is encoded by its first and last sublabels which are 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 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) then the first signature is stored (*Storing phase*, selecting one of its sublabels) and the next one is processed in the *Start phase*.

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 signature by the actual sublabel then it is a compressible one. The successor of the actual sublabel becomes the new actual sublabel, the following 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 compressed 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 is increased by one, otherwise the actual pair is stored in the next element of the compression buffer.

Figure 2. SEIS encoding of a CFG and compression of a signature sequence

3.3. Limitations of the SEIS compression

The construction of the original 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. The efficiency of the compression can be improved if *path optimization* is performed: nodes belonging to frequently executed paths are encoded by successive compressible signatures. A run-time sequence of compressible signatures is broken at a branch statement if not the preferred path is selected, thus the efficiency of the compression depends on the prediction of the branch selection in the signature assignment phase.

Path optimization is performed by executing transformations on the CFG before the assignment of the sublabel values, still preserving the structural properties (i.e. not introducing additional paths). The following transformations are defined:

- Shuffling the input or output edges of a node, i.e. reversing the endpoint or start-point sublabels of the edges in the corresponding signature.
- Introducing duplicated edges between nodes of the CFG.

The first transformations produce compressible signatures in a given path, the second one (which can be followed by the first ones) enables a signature to be embedded in several different signature sequences.

The actual implementation of the SEIS encoding limits the number of sublabels in a signature to 3. Consequently, a signature can be included in maximum 3 different compressible run-time sequences. Additionally, the order of input/output edges in nodes belonging to special statements (exceptional cases in the structural languages, like `goto`, `break` etc.) is further constrained (discussed in details in [8]), resulting in the fact that these nodes usually terminate the compressible signature sequences.

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. However, the lack of dictionary makes the SEIS compression still attractive.

4. Measurement results

The real-time signature compressor was built using an FPGA circuit (Xilinx XC3064 series, 224 configurable logic block each with 2 flip-flops) which needs only an interface to receive signatures and a memory array to store the compression buffer (and the dictionary, in the general case). The compression algorithms are implemented by state machines (a few dozen of states). 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 a conventional watchdog-processor unit [9].

4.1. Size of the compressed trace

The efficiency of the two algorithms was demonstrated first by simulation, compressing the entire run-time signature sequence of different benchmark programs (to compute the size of the buffer necessary to store the entire sequence). In the first scheme, the storage required to keep the sequence could be reduced to 10-30% by compression, depending on the size of the dictionary. The second algorithm provides similar results, but the improvement of the compression rate is difficult (branch prediction in the preprocessing phase).

The effectiveness of the compression based on dictionaries 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 constructing 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). The results are presented in Table 1. The encoded paths inside the iteration loops of the solver are relatively small, thus the compression rate is sensitive to small changes in the dictionary. The data dependency is demonstrated by starting the solver with different parameters (number of levels).

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

Table 1. Size of the compressed trace using dictionaries

The effectiveness of the SEIS compression depends on the structure of the CFG. The measurements were made using various benchmark programs without additional path optimization. The results are satisfactory even in this case (Table 2). Signature sequences of benchmarks with relatively small iteration loops are compressed efficiently, nested loops and complex control structures make the compression difficult.

4.2. Utilization of the compression buffer

The utilization of the circular compression buffer of the diagnostic WP was measured (in each storage cycle, the number of signatures in the buffer was derived and then averaged). The buffer was able to store 256 elements, the counter part of each buffer element was stored in 16 bits. In this prototype, the utilization of the buffer was above 300% in both cases. The results of the SEIS compression of the previous benchmarks are in the last two columns of Table 2. Additionally, the time function of the number of signatures in the buffer is presented for the multigrid benchmark. It reflects the structure of the program, iteration cycles result in peaks in the buffer utilization.

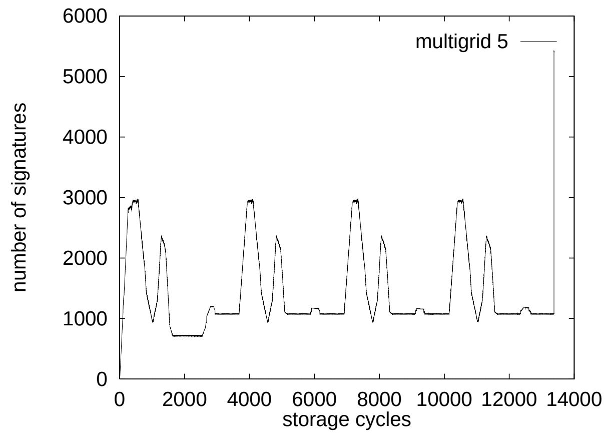

Figure 3. Number of signatures in the compression buffer (SEIS method).

5. Support of debugging

The compression buffer stores a limited number of signatures in a compacted form. If a trigger condition is detected (e.g. an error by the WP) then the execution of the program is stopped and the compression buffer can be accessed by the checked computer itself or by external devices as part of the diagnostic/debugging procedure. Using the decompressed signature sequence, the trace of statements executed before the trigger event can be derived and analyzed.

The debugging of programs can be further supported. First, if the trigger event is reproducible then the dictionary can be reconstructed or a path optimization can be

Benchmark	Number of run-time signatures	Size of the compressed trace	Compression rate	Average buffer utilization	Max. buffer utilization
multigrid 3	3,993	1,008	0.25	320%	485%
multigrid 5	79,005	13,386	0.17	554%	2118%
multigrid 7	1,254,821	160,073	0.13	742%	26,441%
whetstone	118,793	38,895	0.33	302%	4,698%
dhrystone 100	12,288	3,705	0.30	319%	351%
linpack	11,825,895	603,300	0.05	1,960%	10,142%

Table 2. Compression results using the SEIS method

performed on the basis of the contents of the compression buffer, in this way a longer signature sequence can be stored. Additionally, 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 monitored 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.

Our environment can help the input-domain based test of programs. 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 and future work

In our paper a new approach of signature-based monitoring and debugging of programs is proposed. It is shown that the trace based monitoring can be performed using the very same software (program preprocessor) and hardware (signature monitor) as for one of the concurrent error detection techniques, the application of watchdog processors. 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. Both schemes support the implementation of very simple hardware compressor modules which are comparable in complexity with the signature evaluation module of the WP.

Our future work is concentrated on the refinement of the diagnostic environment and on the improvement of the user-controlled dictionary construction and path optimization (in the case of SEIS signatures).

References

[1] D. Bhatt, A. Ghonami, and R. Ramanujan. An instrumented testbed for real-time distributed systems development. In *Proc. IEEE Symposium on Real-Time Systems*, pages 241–250, 1987.

[2] J. P. Calvez and O. Pasquier. Real-time behavior monitoring for multi-processor systems. *Microprocessing and Microprogramming*, 38:213–220, 1993.

[3] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, 1990.

[4] C. R. Hill. A real-time microprocessor debugging technique. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, pages 145–148, 1983.

[5] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 31:681–685, 1982.

[6] E. Maehle and W. Obeloer. Delta-T: A user-transparent software monitoring tool for multi-transputer systems. *Microprocessing and Microprogramming*, 35:245–252, 1992.

[7] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors: A survey. *IEEE Transactions on Computers*, 37:160–174, 1988.

[8] I. Majzik. SEIS: A program control flow graph encoding algorithm for control flow checking. Technical Report TUB-TR-94-EE14, Technical University of Budapest, 1994.

[9] I. Majzik, A. Pataricza, M. D. Cin, W. Hohl, J. Hoenig, and V. Sieh. Hierarchical checking of multiprocessors using watchdog processors. In K. Echtle, D. Hammer, and D. Powell, editors, *Dependable Computing - EDCC-1*, volume 852 of *LNCS*, pages 386–403. Springer Verlag, 1994.

[10] E. Michel and W. Hohl. Concurrent error detection using watchdog processors in the multiprocessor system MEMSY. In *Fault Tolerant Computing Systems*, number 283 in Informatik Fachberichte, pages 54–64. Springer Verlag Berlin, 1991.

[11] A. Pataricza, I. Majzik, W. Hohl, and J. Hoenig. Watchdog processors in parallel systems. *Microprocessing and Microprogramming*, 39:69–74, 1993.

[12] H. Tokuda, M. Kotera, and C. W. Mercer. A real-time monitor for a distributed real-time operating system. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 68–71, 1988.