A High-speed Watchdog Processor
for Multitasking Systems

I. Majzik[†], A. Pataricza[†,‡], W. Hohl[‡], J. Hönig[‡], V. Sieh[‡]

[†] Department of Measurement	[‡] Department of Computer
and Instrumentation Engineering	Science III
Technical University of Budapest	University of Erlangen-Nürnberg
Müegyetem rkp. 9	Martensstr. 3
H-1521 Budapest, Hungary	D-91058 Erlangen, Germany
email: majzik@mmt.bme.hu	email: hohl@informatik.uni-erlangen.de

Abstract

A new watchdog processor scheme for concurrent checking of program control flow is presented. This method is intended to check state of the art processor architectures with on-chip caches as building blocks of multiprocessor systems. The signatures are assigned, so that the processor instruction bus needs not be monitored. The run-time and reference signatures are embedded into the checked program, making obsolete the reference database and the time-consuming search and compare engine in the watchdog processor.The scheme is extended to check multitasking and multiprocessor systems.

1. Introduction

Since computing intensive applications require extremely long execution times, fault tolerance in microprocessor systems becomes a key design factor. As it was shown by both previous practical experience and by fault injection experiments, the majority of computer failures originates in transient faults. A high portion of these faults is manifested as disturbances in the program control flow [1]. An efficient, concurrent check of the control flow has become a basic requirement for the fault-tolerant operation of the computer system.

The most widely used approach to concurrent checking of the program control flow is the application of signature-based watchdog processors (WP) [2]. A WP is a relatively simple coprocessor that compares the actual control flow - represented by run-time signatures - and the previously computed reference control flow. If a mismatch is detected, fault handling in the checked system is activated.

WP methods can be grouped according to the run-time signature generation and the method used to store the reference control flow.

The methods using *derived run-time signatures* monitor the state of the processor using compact abstractions of its behavior (e.g. machine code, control lines). However, these methods require observing the instruction bus, which is not possible in the case of on-chip memory or cache. Asynchronous Signature Instruction Streams (ASIS [3]) and Roving Monitoring Processor (RMP [4]) methods evaluate the signatures using a WP-internal signature database. The Watchdog Direct Processing (WDP [5]) method uses a special WP program containing

the reference signatures and simple instruction codes according to the possible branch instructions of the main program. Basic Path Signature Analysis (BPSA [6]) and its improved versions use reference signatures embedded into the main program (e.g. justifying signatures).

If the instruction bus of the checked processor is not observable, only the *assigned signatures* (AS) method can be applied. The flow of high level language instructions is checked as follows: signatures are assigned to each high level instruction and explicitly transferred to the WP by the main program. A preprocessor extracts from the source code of the user program the control structure in the form of a control flow graph (CFG). In the simplest case, the vertices of the CFG correspond to the statements in the user program and directed edges describe their potential execution order. The preprocessor labels CFG vertices (instructions) with signatures and inserts operations to transfer them to the WP into the source code. During the main program run these signatures uniquely identify the main program location for the WP. The WP receives and evaluates them concurrently. The signature sequence is accepted as correct if it corresponds to a syntactically existing path in the CFG, independently of the semantic correctness of the branch selections.

In the first published assigned signature method Signature Integrity Checking (SIC [7]) the reference control flow is represented by a WP program consisting of signature receive and compare instructions. In the Extended Signature Integrity Checking (ESIC [8]) method the CFG is decomposed into a set of subgraphs each corresponding to a procedure. The reference program control flow is represented as a database defining the valid successors of each signature by a set of adjacency matrices of the CFG subgraphs. Before the start of the main program the signature database is downloaded to the WP, the procedure calls are checked by a finite deterministic stack automaton. The WP has to be implemented by a general purpose microcomputer since the search and compare operations require high computing power, programmability and a large reference memory. Due to this complexity the signature evaluation is slow.

After a brief review of the basic definitions, Section 2 describes a new signature assignment algorithm. The extraction and encoding algorithm of the CFG are presented for programs written in C (Section 3). The signature evaluation is extended in order to support hierarchical checking of multitasking environments. Section 4 describes the aspects of the checking hierarchy. In Section 5 characteristic measurement results of first experiments in the MEMSY multiprocessor are presented. Section 6 is a brief conclusion.

2. The SEIS program control-flow graph encoding algorithm

The main goal of our method called *Signature Encoded Instruction Stream* (SEIS [9]) is to avoid the most important drawbacks of the assigned signature methods, i.e. SEIS intends to allow high-speed signature evaluation and to eliminate the large-scale reference database. In SEIS each signature identifies both the *actual state of the main program* and *the valid successor signatures* as well. The actual run-time signatures can be evaluated using only the previous valid signature as a reference. The fast and simple evaluation scheme and the absence of the reference database supports the use of this method in multitasking systems as well.

In high level programs the number of predecessors and successors of a program state is small for the overwhelming majority of states. In the case of limited number of successors, a state can be identified as the concatenation of the labels of the successors. The SEIS encoding algorithm is a sophisticated implementation of this simple idea.

In the first step of the SEIS method (as common in assigned signature methods) a preprocessor extracts the CFG from the source code. In the second step, often referred to as the *encoding of the CFG*, the signature assignment and `send_signature` command insertion

is performed. Finally the preprocessed (encoded) application program is compiled to the machine code of the checked processor. After starting the program the processor transfers the signatures identifying its state to the WP which then checks concurrently the correct order of the received signatures.

2.1 Basic definitions

Before presenting our method, some basic definitions are recalled:

- The *program control-flow graph* of a program is a couple *(V,E)*, with *V={v_i: i=1,2,...,|V|)}* the set of vertices, each representing a set of succeeding statements of the program, and with *E⊆V×V* the set of directed edges representing syntactically allowed control transfer operations.
- *Input (output) edges* of a vertex v_i are the edges directed to (from) it. The number of input (output) edges is referred as *incoming (outgoing) degree* of v_i: *$deg^-(v_i)$* and *$deg^+(v_i)$*, respectively. A vertex v_j is a *predecessor* of vertex v_i if there is a directed edge *(v_j,v_i)∈E*. Similarly, vertex v_k is *successor* of vertex v_i if *(v_i,v_k)∈E*. A vertex without predecessors is an *initial vertex* of the graph, the vertex without successors is a *final* vertex. Each CFG has at least one initial and one final vertex.
- The *complexity $C(v_i)$* of a vertex v_i is the maximum of the numbers of its predecessor and successor vertices: *$C(v_i)=max\{deg^-(v_i), deg^+(v_i)\}$*.

The vertices of the CFG are encoded by an injective function f_v: *V→S* where *S* is the set of the signatures. A signature s_j is *valid successor* of s_i corresponding to the CFG *G=(V,E)* if and only if *∃(v_i,v_j)∈E* that *$f_v(v_i)=s_i$* and *$f_v(v_j)=s_j$*.

Let us also recapitulate a definition and a theorem of the basic graph theory:

- A directed graph *$G_E=(V_E,E_E)$* is called a *directed Euler-graph* if *∀n_i∈N_E*: *$deg^+(n_i)=deg^-(n_i)$*.
- Theorem: A connected, directed graph is a directed Euler-graph if and only if there exists a closed directed edge trail, the so-called *directed Eulerian circuit*, containing all edges in the graph exactly once.

2.2 Basic encoding of the program control-flow graph

The checking of the validity of the control flow requires the extraction of the directed edge sequences from the CFG, which can be reduced to the well-known problem of the Eulerian circuit generation.

To this end, the CFG has to be transformed into a directed Euler-graph by inserting additional edges in order to eliminate in each vertex the potential difference between the cardinalities of incoming and outgoing edges. In order to get the desired directed Euler-graph, at each vertex v_i, $C(v_i)$-$deg^+(v_i)$ output edges and $C(v_i)$-$deg^-(v_i)$ input edges have to be inserted. The Eulerian circuit can now be generated using effective algorithms [10]. The simplest, the Hierholzer-algorithm, for example, generates a maximal circuit starting from a given vertex, then generates circuits from each participating vertex and inserts them into the current one until no new vertices remain.

The main idea of the *Basic Encoding Algorithm* proceeds as follows (see the example presented in Figure 1). The deletion of one of the additional edges in the Eulerian circuit results in a directed edge trail *T*. The vertices of *T* are assigned code values called *sublabels*. Since the Eulerian circuit contains each v_i∈V exactly $C(v_i)$ times, each vertex signature will consist of the concatenation of the $C(v_i)$ sublabels. The sublabels are ordered to the vertices of *T* in the

1. Transformation of the CFG:

An example C program:

```
a;
if (?) {b; if (?) goto lc;}
else for (;;) {
lc:    c;
        if (?) break;
        d;}
e;
```

Program control-flow graph:

The directed Euler-graph:

→ Additional edge

2. A directed Eulerian circuit:

$a_1 \to b_1 \to e_1 \to b_2 \to c_1 \to a_2 \to c_2 \to d \to c_3 \to e_2$

4. Encoding of the edge trail: e.g. $l_0 = 1$, $f_s(l_i) = l_i + 1$

$1 \to 2 \to 3 \to 5 \to 6 \to 8 \to 9 \to 10 \to 11 \to 12$

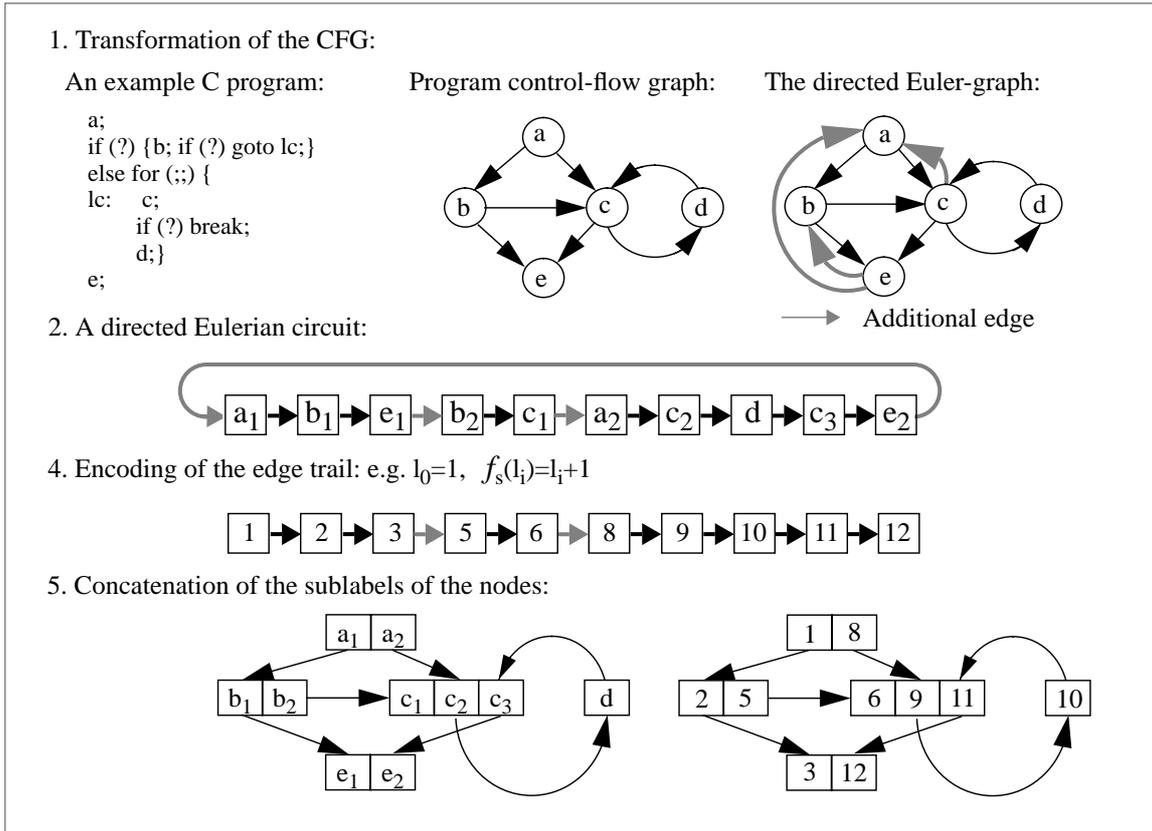**5. Concatenation of the sublabels of the nodes:**

Figure 1   Encoding of a program control flow graph

following way:

The set of sublabels is ordered to form a sequence $L=(l_0, l_1, \dots, l_M)$ defining a successor function $f_s\colon L \to L$. The first vertex of T is assigned the initial sublabel l_0, the next vertices are assigned subsequently. If the next vertex is connected by a normal edge then it is assigned the successor of the previous sublabel. When encountering an additional edge existing only in the Euler-graph, but not in the CFG, a sublabel value is skipped and so the vertices will be separated. A sublabel l_j is called *code-successor* of sublabel l_i if $l_j = f_s(l_i)$, i.e. they are not separated.

The signature evaluation can be reduced to the evaluation of the sublabels. A signature s_j is valid successor of s_i if and only if one of the sublabels of s_j is a code-successor of one of the sublabels of s_i. The algorithm described above satisfies this requirement as it can be proved based on the following facts:

- the Eulerian circuit contains each edge of the CFG once and exactly once;
- the endpoint sublabels of normal edges are code-successors while the endpoints of the additional edges are separated by an unused sublabel value;
- the f_s function is increasing strictly monotonic in respect to the selected ordering (each sublabel has an unique value).

2.3 The SEIS encoding of the program control flow graph

The Basic Encoding Algorithm described above performs a variable length encoding of the vertices of the CFG. From the technical point of view a fixed length encoding would better support the requirement for a high speed evaluation and fixed word length in the signature transfer. The required length (i.e. the number of sublabels) corresponds to the complexity of the vertices in the CFG.

Elementary control structures of a programming language can be categorized (resulting from the syntax of the language) on the basis of the CFG complexity limitations as follows:

- *Normal statements* can be represented by a control-flow subgraph of *limited complexity* of its component vertices, generally $C(v_i) \leq 2$. They incorporate simple control structures like value assignment, `while` type iteration, `if...then...else` conditional branches etc.

- *Multiple output statements*, like a `case` statement can have an *arbitrary number of successors*. Such statements are represented in the CFG by *multiple output vertices* with an arbitrary number of output edges. Their successor statements will be referred as *special successors*. The vertices representing special successor statements are *special successor vertices*, their input edges are *special successor edges*.

- *Multiple input statements* can have an arbitrary number of predecessors, like a labelled instruction to which `goto` statements are directed. Such a statement is represented by a *multiple input vertex*. Their predecessors will be referred as *special predecessor*s. The vertices in the CFG representing special predecessor statements are *special predecessor vertices*, their output edges are *special predecessor edges*.

Normal statements have a limited complexity of a typical value of one or two. Multiple input and multiple output vertices, however, have to be specially encoded in order to reduce their number of sublabels. The basic idea of this reduction is that sublabels referring to the same successor (predecessor) signature can have the same value if they have no predecessor (successor) sublabels. In this case their output (input) edges can point to the same sublabel reducing the number of sublabels of the successor (predecessor) signature. The SEIS encoding method takes the advantage of this possibility: multiple sublabels which are separated from their predecessors (successors) by inserting an additional self-loop in the corresponding vertices get the same successor (predecessor) sublabel. The SEIS encoding algorithm is given formally as follows:

SEIS Control Flow Graph Encoding: Given a control-flow graph, the result is a statement label assignment. The complexity of the multiple input and multiple output vertices is reduced.

   Step 1. Mark the special predecessor, special successor, multiple input and multiple output vertices, according to the type of the represented statements.

   Step 2. Insert $|E_p|$ additional self-loops in special predecessor vertices, where $|E_p|$ denotes the number of special predecessor edges of the vertex. Similarly, insert $|E_s|$ additional self-loops in special successor vertices, where $|E_s|$ denotes the number of special successor edges of the vertex.

   Step 3. Transform the CFG into an Euler-graph inserting the necessary additional edges.

   Step 4. Generate an Eulerian circuit. The Hierholzer-algorithm can be applied with a slight modification: the selection of the next edge during the circuit generation and the insertion of a circuit into the maximal circuit, respectively, should be determined by the following constraints:

   - in special predecessor vertices the self-loops have to be succeeded by special predecessor edges;
   - in special successor vertices the special successor edges have to be succeeded by additional self-loops.

   The reduction of sublabels will be efficient if at multiple input (multiple output) vertices the special predecessor (special successor) edges are succeeded by (succeed) additional edges.

Step 5. Reduce the edge sequences consisting of additional edges replacing them by a single additional edge connecting the start point of the first additional edge with the endpoint of the last additional edge.

Step 6. Delete an additional edge and encode the resulting edge trail by the Basic Encoding Algorithm.

Step 7. Reduce the complexity of the multiple input and multiple output vertices:
In each multiple input vertex collect those sublabels into the set L_p which are endpoints of special predecessor edges. Select a sublabel l_p from L_p (if it is possible, a sublabel which is start point of a normal edge as well) and reassign the $f_s^{-1}(l_p)$ value to the startpoint sublabels of <u>all</u> edges whose endpoint is in $L_p\text{-}\{l_p\}$. Delete the sublabels in $L_p\text{-}\{l_p\}$ which are start points of additional edges.
Similarly, in each multiple output vertex collect those sublabels into the set L_s which are startpoints of special successor edges. Select a sublabel l_s from L_s (if it is possible, a sublabel which is endpoint of a normal edge as well) and reassign the $f_s(l_s)$ value to the endpoint sublabels of <u>all</u> edges whose start point is in $L_p\text{-}\{l_p\}$. Delete the sublabels in $L_p\text{-}\{l_p\}$ which are endpoints of additional edges.

Step 8. Append $C\text{-}|L(v)|$ new sublabels to the statement labels by simply duplicating one of the existing sublabels (C is the predefined complexity of the CFG which can be guaranteed by the algorithm, $|L(v)|$ is the actual number of sublabels of v). In this way at the end of the encoding each statement label consist of C sublabels.

A general CFG can be encoded using the SEIS algorithm, complying to the requirement that each signature consists of C sublabels, if the conditions presented in Table 1 are satisfied. A detailed description and analysis is found in [11].

Vertex/edge type	Normal input	Normal output	Special successor	Special predecessor
Normal	C	C	-	-
Special predecessor	$C\text{-}|E_p|$	-	-	$|E_p|\leq C$
Special successor	-	$C\text{-}|E_s|$	$|E_s|\leq C$	-
Multiple input	$C\text{-}1$	C	-	*unlimited*
Multiple output	C	$C\text{-}1$	*unlimited*	-

Table 1   Maximum number of edges depending on the C complexity limit

3. Encoding of C programs

The SEIS encoding algorithm for C programs is implemented in the preprocessor. The first step of the encoding is the graph extraction: Statements of the program have to be defined as vertices in the CFG.

The C program will be decomposed into separate control flow subgraphs according to its procedural structure. Within a procedure, some statements can be merged into a compound statement and can be used as a single statement in other control structures. This forms the base of the hierarchical graph composition: Elementary control subgraphs having initial and final vertices can be assigned to the (compound) statements and these subgraphs can be used as single vertices in higher-level graphs connecting the input edges to the initial vertex, the output edges from the final vertex.

Most control structures of the C language can be mapped to subgraphs of $C(G)=2$ complexity (Figure 2). Since initial vertices have no input edges, final vertices have no output
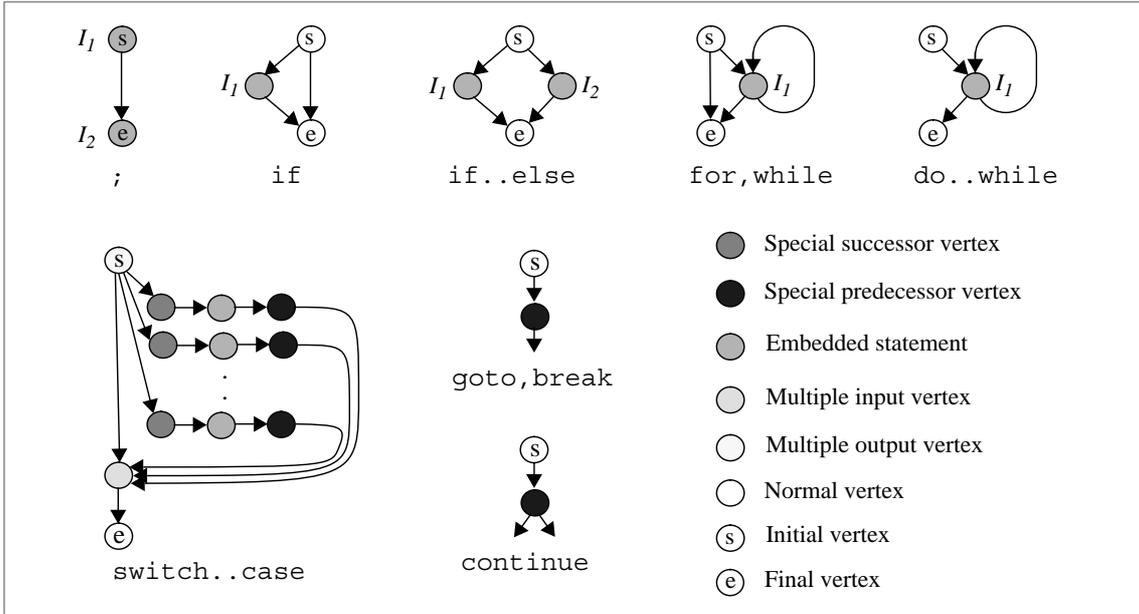
Figure 2   Typical control structures of the C language

edges and internal vertices are of complexity $C(v)=2$, these subgraphs can be composed complying with the $C(G)=2$ requirement. The remaining other control structures are represented in the way presented in Figure 2: `case` branches as special successor statements are represented by special successor vertices, the `switch` statement is mapped to a single multiple output vertex; `goto`, `break` and `continue` statements as special predecessors are represented by a subgraph consisting of a normal and a succeeding special predecessor vertex.

The CFG built from these building blocks satisfies the requirements of the SEIS encoding algorithm using as a maximum *3* sublabels per vertex (check Table 1 for *C=3*).

4. Aspects of hierarchical checking

The SEIS WP is intended to be used in multitasking, multiple processor systems. Therefore, the signatures contain additional information: identification of the procedure, task and processor in order to extend the checking. In our method a signature corresponding to a vertex consists of the following fields (Figure 3):
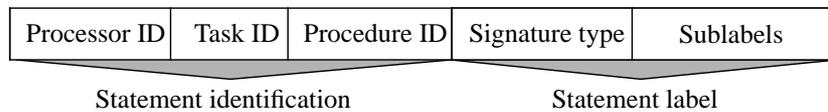

Figure 3   Signature structure

The main advantage of the SEIS signature assignment is that the control-flow checking requires only very limited hardware resources. The block diagram of a statement label checker is presented in Figure 4 (for simplicity 2 sublabels per signature are used).

Based on the encoded statement ID (Figure 3), hierarchical checking of the program control flow is possible:

•   *Statement level*: Statements are labelled by signatures consisting of a fixed number of sublabels. The evaluation of a signature needs only a single reference signature which is identical with the last valid one. When receiving a new actual signature, the sublabels of
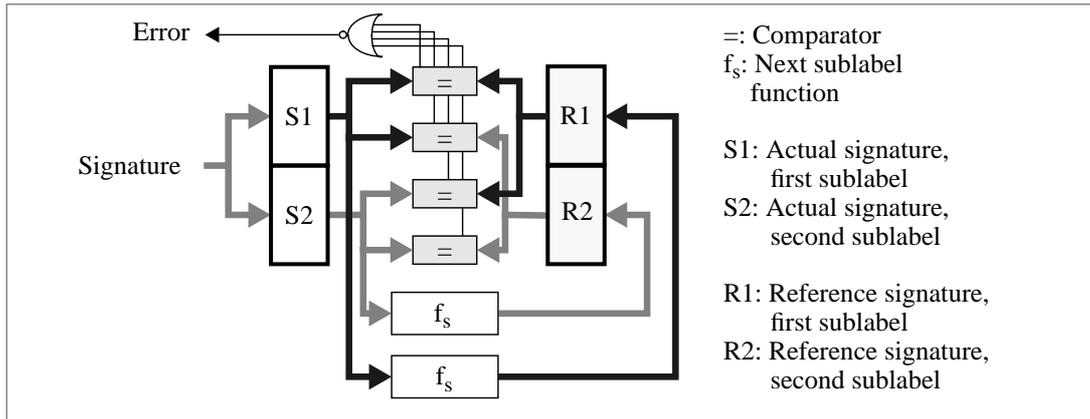
Figure 4   A simple statement label checker

the reference signature registers of the WP are modified by the sublabel successor function f_s and then compared with the sublabels of the actual signature. If any pair of sublabels is identical, the actual signature is valid.

- *Procedure level*: Each procedure is mapped to a control-flow subgraph, which can be checked independently. The procedure calls are handled by the SEIS WP in an identical way as in the ESIC method. The initial and final vertices of the CFGs of each procedure are marked by special *start of procedure* (SOP) and *end of procedure* (EOP) flags (encoded in the signature type field). Receiving a SOP signature the WP stores the reference signature of the caller procedure on a *signature stack* and checks the actual procedure using the SOP signature as a first reference. Receiving an EOP, after the evaluation of it, the old reference signature corresponding to the calling procedure is restored from the stack and used as the new reference. The procedures are identified by their procedure ID embedded in the signature. The procedure ID may change only in the case of a SOP signature.

- *Task level*: Separate signature stacks in the WP are assigned to the different tasks. Each stack stores the reference signatures of the procedure calls of the task; on the top the actual reference is found. On the base of the task ID embedded in the signature the WP can switch over to the corresponding stack: the context switch is a simple stack pointer selection. Since the task ID is transferred to the WP embedded in the signature address word, it can be determined in run-time using the virtual to physical address translation capability of the MMU of the checked processors.

  Using the processor and the task identifier, the task scheduler can be checked as well. The WP stores the ID of the task actually executed by a given processor in a *processor-task database*. If a signature is received, the actual and the reference task ID are compared (concurrently with sublabel comparison). Upon scheduling a new task on a given processor the processor-task database has to be updated.

  Additionally, the WP contains a *signature transfer timer*. If a task fails to transfer signatures to the WP within a given interval, the timer signals a time-out. The timer is controlled by the processor-task database: only the presently running task is checked.

On the base of the hierarchical checking the WP can signal a control flow error as wrong statement label, wrong procedure ID, wrong task ID, or signature transfer time-out. *Error recovery* is supported by checkpoint databases (stored signature stack spaces) generated in the WP for the running tasks. The WP-internal checkpoint storage and backward recovery can be initiated by special commands.

5. Measurement results

The experimental version of the SEIS WP was implemented in the MEMSY (Modular Expandable Multiprocessor System) [12]. An elementary pyramid consisting of five processing nodes is checked by a single shared watchdog-processor to reduce the hardware overhead and to estimate the consequences of the multiprocessing environment for the design of the WP [13].

Multiple benchmarks were selected for the validation of error coverage and run time overhead. In the following, results will be presented for the most characteristic benchmark, a multigrid differential equation solver. For overhead validation the following strategies were selected:

- SEIS encoding in the form in which it was described earlier.
- Static reduction of signatures: subsequent statements with a single input and a single output in the original program are merged into a single vertex in the CFG.
- Dynamic reduction of signatures: In short iteration loops, only every n-th signature is transferred to the WP.

Fault coverage was measured using fault injection experiments. The fault set to be injected was selected in such a way, that triggering of the primary error detection mechanisms (like segmentation violation or false bus alignment checks) is avoided in order to eliminate interference effects between different approaches as far as possible.

As illustrated in Figure 5, a relative moderate reduction in the maximal frequency of the signatures reduces the run time overhead from 60% (in other applications even several hundreds of percents) to a typical value of 10-15%. However, the fault coverage rapidly decreases using intensive signature reduction.
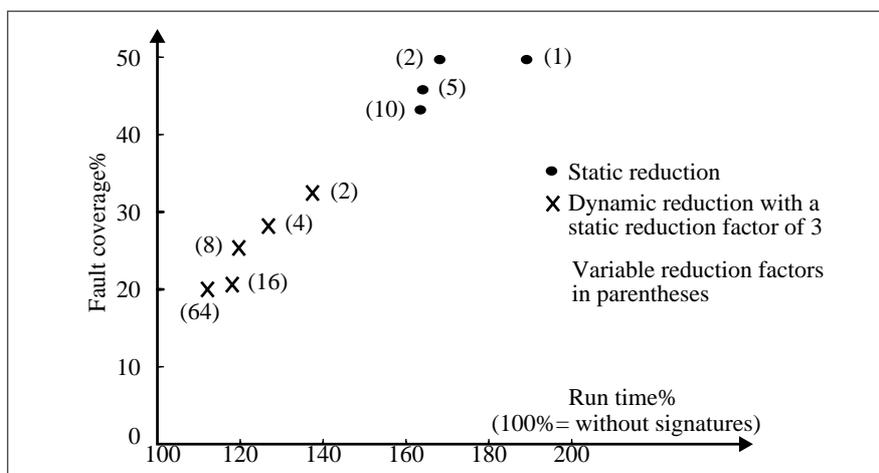

Figure 5   Run time vs fault coverage

6. Conclusions

In the paper a new assigned signature based watchdog processor is presented, which is intended to check multitasking systems. From the experiments, the following applicability conditions of the SEIS method can be derived:

- The SEIS method performs the fastest signature evaluation known among the different AS methods. Its signature checker hardware is simple, the checks can be extended to higher levels of the application using additional checker modules. The preprocessor-approach assures a portable and compiler-independent signature assignment. However, as for all AS methods, existing programs which can not be recompiled, can not be checked.

- A proper compromise has to be found between the fault coverage and the applied signature reduction (run-time overhead). Preprocessing higher level languages, the implicit assumption on the relation between signature transfer frequency and number of statements between consecutive signatures is only a very rough estimation.

- If the signature transfer is slow (comparing with the memory access cycle in the system) then the time overhead of the preprocessed program will be unacceptable high, even if the signature evaluation is fast. If the main processor uses speed-up mechanisms like instruction prefetch queue and cache, a slow signature transfer becomes a performance bottleneck in the checked system.

References

[1]	Gunneflo, U.; Karlsson, J. and Torin, J.: "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", *Proc. FTCS-19*, pp. 340-347, 1989

[2]	Mahmood, A. and McCluskey, E. J.: "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Trans. on Comp., Vol 37/2*, pp. 160-174, 1988

[3]	Eifert, J. B. and Shen, J. P.: "Processor Monitoring Using Asynchronous Signatured Instruction Streams", *Proc. FTCS-14*, pp. 394-399, 1984

[4]	Shen, J. P. and Tomas, S. P.: "A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems", *Microprocessing and Microprogramming 20*, pp. 249-269, North-Holland, 1987

[5]	Michel, T.; Leveugle, R. and Saucier, G.: "A New Approach to Control Flow Checking Without Program Modification", *Proc. FTCS-21*, pp. 334-341, 1991

[6]	Sridhar, T. and Thatte, S. M.: "Concurrent Checking of Program Flow in VLSI Processors", *Proc. 1982 Int. Test Conf.*, pp. 191-199, 1982

[7]	Lu, D. J.: "Watchdog Processors and Structural Integrity Checking", *IEEE Trans. on Comp., Vol 31/7*, pp. 681-685, 1982

[8]	Michel, E. and Hohl, W.: "Concurrent Error Detection Using Watchdog Processors in the Multiprocessor System MEMSY", *Informatik-Fachberichte 283, Fault Tolerant Computing Systems*, pp. 54-64, Springer Verlag Berlin, 1991

[9]	Pataricza, A.; Majzik, I.; Hohl, W. and Hönig, J.: "Watchdog Processors in Parallel Systems", *EUROMICRO'93, 19th Symposium on Microprocessing and Microprogramming*, Barcelona, 1993

[10]	Gould, R.: "Graph Theory", The Benjamin/Cummings Publishing Company, Inc., 1988.

[11]	Majzik, I.: "SEIS: A Program Control Flow Graph Encoding Algorithm for Control Flow Checking", *Internal report*, Technical University of Budapest, 1994 (in press)

[12]	Dal Cin, M. et al.: "Fault Tolerance in Distributed Shared Memory Multiprocessors", *in: A. Bode, M. Dal Cin (eds.): Parallel Computer Architectures - Theory, Hardware, Software, Applications*, Springer Lecture Notes in Computer Sciences 732, 1993, pp. 31-48

[13]	Majzik, I.: "Fault Detection in the MEMSY Multiprocessor using a SEIS Watchdog Processor", *Internal report 10/1993 of the IMMD3*, Universität Erlangen-Nürnberg, 1993