Chapter 4

Concurrent error detection

Fault simulation, fault injection experiments and analysis of error logs of operational systems show that the majority of errors in microprocessor systems is caused by transient faults. This way the fault-tolerant operation of the system requires efficient concurrent error detection techniques. The most widely used technique of hardware redundancy based concurrent error detection is the application of run-time monitors. They monitor the state and operation of the supervised system, comparing the run-time information with the reference one corresponding to the fault-free operation of the system. In case of a difference, an error is signaled. The run-time information is often represented in a compact form, by signatures (symbolic labels).

Transient faults in microprocessors often manifest in disturbances in the control flow. Thus run-time monitors intended to check the control flow of the microprocessor are especially important. Moreover, also the software based checking techniques, like reasonableness checks, require the assurance that the checking routines are executed and not omitted as a consequence of the erroneous behavior. One type of that monitors are watchdog processors (WP) which concurrently check the control flow, i.e. the order of the instructions executed by the processor, using signatures, compact abstractions of the system state.

This chapter presents the concurrent error detection methods commonly applied in (multi)-processor systems. The most important method, the application of watchdog processors is described in details.

4.1 Basics of error detection

The task of the error detection mechanisms is to find out whether the actual state of the system is erroneous. The original system is supplemented by additional components which perform the check. *Ideal checks* (i) are based solely on the specification, (ii) check the complete behavior of the system and (iii) are independent of the checked system (to avoid the effects of errors to the checks) [LA90]. The incomplete specification, the complexity of the system and the need of accessing the information to be checked usually prevents the implementation of an ideal checker. *Acceptability checks*, which are less strict, mean that the absence of errors is not guaranteed, but the results can still be accepted with a high probability.

The main characteristics of error detection are the overhead required to implement the check, the error latency and coverage. Low latency is especially important to avoid the spread of error effects and damage of critical system components (e.g. recovery data). A long error latency can also prevent the fault diagnosis by the weak correlation between the fault and its symptoms.

Error detection, as well as fault tolerant operation of the system, is based on some form of redundancy, i.e. utilization of extra elements. System redundancy may include

- structural (or hardware) redundancy: presence of extra hardware components (e.g. replication of the processor),
- functional (or software) redundancy: presence of additional functions (e.g. N-version programming),
- information redundancy: complementary information added to the original one (e.g. checksum, parity bits, error detection and correction codes),
- time redundancy: additional time which is needed to deliver the service of the system (e.g. multiple execution of the operation).

Error detection can be performed off-line, when the system is inactive (or is in idle state) or on-line, when the system is operational. The latter is referred to as *concurrent* error detection. Note that detection of transient faults requires continuous, concurrent error detection. Off-line checks usually include the systematic testing of the system by supplying special inputs (test patterns) to the components and investigating the outputs. The purpose is either to check whether the component satisfies the specification (conformance testing) or to reveal some hypothesized faults (fault testing). This type of checks have high coverage but are expensive in resources and time, this way they are rarely implemented as concurrent diagnostic checks.

The checks used for concurrent error detection highly utilize the redundancies built in into the system. They are categorized in [LA90] as follows:

Replication: The computation of the system is repeated (applying time or structural redundancy), either by the system itself or by an alternative implementation. The replication of the software can detect only transient faults. Examples of hardware replication are master-checker configuration, TMR or NMR structures.

Reversal checks: If there is one-to-one mapping between inputs and outputs of the system, i.e. the inverse of the transfer function exists then the input can be re-calculated and compared to the original one (functional redundancy). Besides mathematical algorithms, the common read-back technique can be mentioned here.

Coding checks: If there is information redundancy in the representation of some objects with a predefined relationship between the original and the redundant representation, then this relationship can be checked. Signature-based control flow checking is also put into this category.

Structural checks: In complex data representations the error-free structure or the built-in redundancy can be examined (e.g. number of elements, pointers). If n changes are required to produce a valid structure, $n-1$ accidental changes can be detected.

Reasonableness checks: Based on the intended usage or purpose of the output or service of the system, some assumptions can be made and checked (e.g. limits, ranges, types of data values).

Timing checks: Timing constraints of the system are examined, e.g. by a watchdog timer. Monitoring based on signals expected to arrive regularly ("I'm alive" messages) can also be mentioned here.

Interface checks: It is checked whether the system components perform only legal operations, and the interaction between the components complies with the specification. Typical errors detected by interface checks are protection violation or illegal operation.

Diagnostic checks: The components of the system are exercised by test patterns. As mentioned above, it is rarely implemented as on-line technique.

To develop and use an error detection method a *fault model* has to be first established and analyzed. It describes the possible faults and their effects (errors) at the appropriate level of the system model. Structural fault models deal with the faults of the various system components, functional fault models describe the possible malfunctions of the system.

4.2 Classification of errors in microprocessor systems

The investigation of the types and effects of faults in microprocessor systems is a topic of continuous research. As the complexity and speed of the devices increases, more and new fault and error types should be taken into account. In most cases, the complexity of the device or the incomplete knowledge of the internal structure and operation prevent the elaboration of a detailed, low level (gate or register transfer level) fault model, this way a meaningful analytical estimation of the effects of low-level faults to the behavior of the system is difficult to achieve. This is particularly true for transient faults, since this class does not have a good characterization. If the internal structure of the device under investigation is available then simulation based techniques can be applied (especially in the design phase), otherwise the experimental evaluation by hardware-based fault injection is a widely used technique (in the prototype phase). The collection of the error log of systems is another source of the analysis (in operational phase).

Some results of the research are presented in the following tables (Table 4.1, Table 4.2, Table 4.3). The overview is focussed on microprocessors and related errors which are relatively easy to detect. The applied techniques include simulation [ORG92], fault injection by heavy-ion radiation [KGT89], power disturbances and modification of signals of the system bus [STDM82], [MQS90]. Software based injection techniques, since they model the effects of faults directly at the error level, are excluded from this survey.

Benchmark	Control flow			Control signal	Data	Other
	permanent	temporary	latent			
Quicksort	62%	8%	16%	8%	4%	2%
Heapsort	45%	6%	24%	19%	5%	1%

Table 4.1: The effects of heavy ion radiation [KGT89]

	Invalid program flow	Invalid opcode address	Unused memory	Invalid read address	Invalid opcode	Invalid write address	Non-existent memory
[STDM82]	63%	59%	56%	44%	37%	19%	8%
[MQS90]		45%	72%		1%	9%	

Table 4.2: Effects of hardware fault injection on the system bus

The first important result of the studies is that transient and intermittent faults are the major causes for digital system failures. It is estimated that they occur 10 to 30 times more frequently than permanent faults, which means that they cause more than 90% of the failures in computers [SS82], [SL81]. Transient faults are caused mainly by power jitter, electro-magnetic interference, ionization by cosmic rays or alpha particles from packaging materials. A minor extent (intermittent faults) is caused by design faults, e.g. fan-in, marginal circuits, oscillation between active and inactive states etc. The decreasing sizes and the high speed of chips make them more sensitive since the difference between the logic levels tends to be lower and the time margins are reduced. In industrial environments where power disturbances and electro-magnetic interference are frequent, transient faults play an even more important role.

As a consequence of the above studies, the most efficient error detection mechanisms are, in addition to the data checking, the checking of control flow, memory access behavior and instruction codes. (The last one is efficient if not all bit patterns are used by the processor as instruction codes.) They are defined at a higher (behavioral) level, abstracted from the underlying gate- or register level structural models.

4.3 Concurrent error detection techniques

As the experiments have shown, a high portion of failures results from transient faults. They are hard to be avoided by careful design (fault avoidance) and testing during the manufacturing

	Control flow error	Data error	Other errors	Bus errors	Address errors	Basic errors
Effect	33%	60%	7%	28%	8%	37%

Table 4.3: Effects of simulation-based fault injection in RISC processors [ORG92]

process (fault removal), thus fault tolerance techniques are needed to improve the dependability of the system. Since the dependability of the system is especially sensitive to undetected or mishandled transient faults [Sos94], the application of efficient error detection techniques is of utmost interest. Moreover, in parallel computing systems running long, computing intensive applications early error detection is a vital requirement [CGH+93], [CP94]. Regularly scheduled off-line tests are usually too time consuming, without being able to detect transient faults, therefore *concurrent, on-line error detectors* are preferred. Since transient faults cause no permanent damage in the system, the chance of a successful recovery is high, especially if the error coverage is high and the latency is low (avoiding spreading the effects of the error and damage of the recovery data).

From the point of view of the level of checking, concurrent error detectors can be divided into three categories: circuit, system and application level techniques. An other categorization is to distinguish checks provided by the "interpreter" (underlying hardware and operating system which maintains an interface on which the application is executed) and checks performed by the interpreted system (application). The general purpose checking facilities provided by the interpreter are referred to as *mechanisms* while the application-dependent ones explicitly programmed in the interpreted system are *measures*.

Circuit level detectors are mechanisms often built in into the system in the design phase when internal data paths and register-level building blocks are accessible, but are also used to detect errors on the external input or output pins of the chips. They are based on coding checks (redundancy of information). Primary inputs and outputs of a module are encoded in an error detection code, and a checker is assigned to highlight the errors (self-checking circuits, [Wak82]). Single words of data are checked e.g. by Hamming, Berger or n-out-of-m codes, data blocks (in transmission subsystems) are checked by cyclic codes. Checking of arithmetic or logic operations requires different type of codes, which are invariant to the operation. A*N codes, residue codes or parity prediction are widely used techniques. Such type of encoding and self-checking are applied to control sections (sequential circuits) too, but the main field of application is the error detection in data sections, where high coverage can be achieved using a small amount of redundancy.

Techniques used at the circuit level can be used within custom-design VLSI chips to provide the low-level detection of errors. However, due to increased circuit density and a reduced access to the internal states of all modules (especially in the control section) the problem to detect errors in the overall behavior of the system is still open.

System level detectors aim at verifying the correctness by checking general properties of the system. They can be considered generally as guardians that check out invalid activities. The usual implementations perform replication checks by applying identical replicas or functionally equivalent variants and some adjudication unit. Hardware redundancy based techniques (including master-checker and other forms of structural replication, even with design diversity) compare the internal or external signals of the checked modules. The former requires tightly synchronized modules, the latter results in higher error latency. Time and software redundancy based replications repeat the computations of the system before producing the output. (Detection of permanent hardware faults requires different activation of the system resources.) Recovery block scheme, N-version programming and their modifications are widely used, differing mainly in the execution of the replicas (parallel, serial, adaptively etc.).

Hardware redundancy based error detection and fault masking are quite expensive, not admissible in low cost systems or in massively parallel systems. Software and time redundancy based techniques often lead to complex synchronization problems and represent high performance loss [Daa86].

As low cost mechanism for hardware-based system level error detection, external monitoring devices are proposed. The hardware and time overhead is reduced as these monitors are relatively simple, since they examine only some selected properties of the system by performing timing, interface and coding checks. Based on the fault injection experiments and practical observations, especially program timing (time-out of some operations), memory access (invalid read or write address, access to unused memory) and control flow (invalid control flow) are proposed as system properties to be monitored. These monitors are generally referred to as watchdog timers (WT) or watchdog processors (WP). They are discussed in details in the following section.

Application level detectors are measures based on properties of the particular algorithm executed by the system. Errors are detected by assertions, invariant relationships among program variables. Usual techniques include reversal, structural and reasonableness checks. Another alternative is monitoring the services to be delivered by the application [TDM+94].

Application level techniques are not automatic and transparent since the system programmer has to analyze the algorithm and derive the relationships and services to be checked. The assertions have to be inserted into the program in some form, the requirements to be checked has to be selected explicitly. These techniques are likely to have high error latency.

Comparing circuit, system and application level techniques we can conclude that the system level techniques become more and more important. First, as the complexity of the circuits grows, the cost/error coverage ratio of the circuit level techniques becomes less satisfactory. Second, using higher level checks, the less detailed elaboration of a fault model (the less knowledge about the detailed internal structure and operation of the system) is needed as more general properties of the system are checked. The error detection latency is acceptable, between the low latency of the circuit level and high latency of the application level techniques. In the category of general purpose systems built upon commercial off-the-shelf components, an efficient on-line error detection can be performed almost only by system level techniques.

4.4 Watchdog processors

A watchdog processor (WP) is a relatively small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of a main processor [Lu80], [MM88]. The general system architecture is shown in Figure 4.1. The WP is provided with some information about the state of the processor or process to be checked on the system (instruction and/or data) bus. Errors detected by the WP are signaled towards the checked processor or any external supervisory unit responsible for error treatment and recovery.

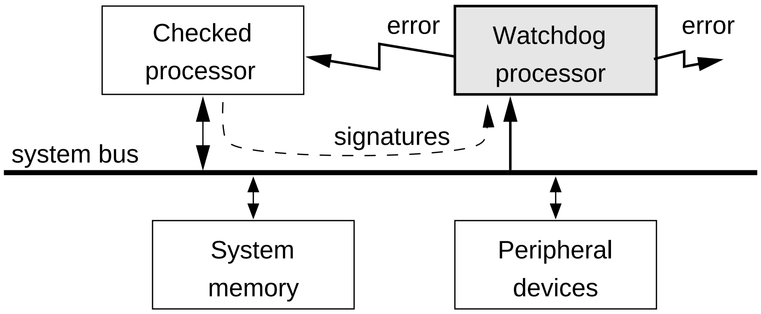

Figure 4.1: General system architecture with WP

From the point of view of information transfer, the operation of a watchdog processor is a two phase process. In the first, initialization phase the WP is provided with *reference information* about the fault-free operation of the checked processor. During the checker phase it is compared to the *run-time information* collected by the WP. In the case of a discrepancy, an error is detected. The general scheme is the one of testing: the WP compares the run-time information from the processor (device under test) with the reference one, the result of the comparison is a signal of *Go/NoGo* type.

The reference and run-time information checked by the WP can be about memory access behavior, control signals, control flow or reasonableness of results. To reduce the information transfer, the reference and run-time information is usually represented in a compacted form, by *signatures* (tokens).

Historically, the WP is an extension of the idea of *watchdog timers* (WT, [CPW72], [OCKB75]). Watchdog timers are simple hardware or software modules used to monitor concurrently the timing of selected system activities. Special reset sequences are embedded into the operation of the system which initialize the WT periodically. If it is not initialized within a specified time then a time-out is detected and an error signal is generated. Hardware implementations are popular in bus systems and communication circuits, software techniques are used e.g. in distributed operating systems or in real-time applications.

The application of watchdog processors as system level error detection technique has several advantages as follows:

- In terms of complexity, the WP is between the circuit level techniques and system level techniques that apply massive redundancy. The application of a WP is simpler and cheaper than the replication of the checked processor.

- A WP can be added to existing systems without major modifications. If the system to be checked uses circuit level error detection techniques, then the WP can increase the error coverage by detecting errors escaping the detection at the lower level.

- Like other system level error detection techniques, the WP does not require simplistic fault models. (It can be compared to functional, off-line testing of processors [TA80].)
- The performance loss in the system is not significant, since the checking is performed concurrently.
- The WP is an independent part in the checked system, this way it is protected against common mode errors (achieving design diversity, [AK84]). (Other schemes use circuits or processors from different manufacturers to overcome this problem.)
- In specific cases, the WP can detect not only hardware errors but also software or design errors by performing concurrent reasonableness checks at the output of the checked processor.
- The WP can support debugging, fault diagnosis and error recovery by storing error-specific informations (e.g. the trace of the instructions executed before the error).
- Application of WPs fits into the trend towards distributed computing by using dedicated (often reconfigurable) coprocessors to perform specialized tasks in the system (calculations, I/O processing). Many systems (e.g. mainframes) use console processors for system maintenance and diagnosis [Avi81, Liu84, CC86], which can be supported by the WP as mentioned earlier.

Designing a concurrent error detection method, mechanisms providing the best error coverage with minimal effort and cost should be selected. A wide variety of watchdog processor methods are proposed by the literature to check the control flow, control signals, memory access behavior and reasonableness of results.

4.4.1 Control flow checking

Correct control flow is a fundamental part of the correct execution of computer programs. It is also a prerequisite for any software-implemented, system or application level error detection technique, since disturbances in the control flow can disable the execution of such checks resulting in undetected errors.

The control flow of a program can be represented by a *control flow graph* (CFG). Nodes of the CFG represent some program units (single statements or blocks of statements) while edges represent the *syntactically allowed* flow of control among them. Often also the *program graph* is defined in which each node represents a single instruction (at assembly level) or a single statement (in higher level languages). In this work, the following notation is used: The application consists of *processes*, a process consists of *procedures*. In high level languages (C, Pascal etc.) a procedure is composed of *statements*. At assembly level, to emphasize the difference, he notation of *instructions* is used.

Control flow checking detects permanent and temporary errors in the control flow. A *permanent control flow error* means that a statement of the program (which was reached by a fault-free path) is followed by a faulty one which does not correspond to the program, and the execution will not return to the original flow of control (program crash). A *temporary control flow error* means that the faulty statement corresponds to the program and the execution continues from this faulty state by an allowed flow of control. As mentioned in Section 4.2, control flow errors are defined at system level, covering a wide range of circuit level faults. Anything that prevents the proper updating of the program counter can result in control flow error: e.g. faults in hardware such as the registers of program counter and stack pointer, program stack, address circuits, microcode memory and faults in software such as compiler bugs, unintentional modification of program code, operating system bugs.

All methods of control flow checking are based on assigning *signatures* (labels, tokens) to the nodes of the CFG. Before the program run, the WP is provided with the signatures labeling the nodes and the relationships among them given by the CFG (reference information). In run-time, the WP monitors the program run and concurrently computes the signatures (run-time information). The run-time signatures are compared to the reference ones provided earlier. If a discrepancy is detected then an error is signaled. In general, the *structure* of the control flow is checked, the details of the control flow related to execution semantics (such as data dependences) are not checked.

By applying signature-based control flow checking, permanent errors are detected by time-out mechanisms (watchdog timer) as the last valid signature is not followed by any signature within a predefined time interval. Temporary errors can be detected by the validation of the signatures as the last valid signature is followed by an invalid one.

Many methods of control flow checking have been proposed. As new processor architectures are introduced, the basic methods are regularly adapted. These methods differ from each other in the definition of a node, in the representation of the reference information and in the derivation of the run-time information. The following categories can be clearly distinguished:

Assigned run-time signatures The signatures labeling the nodes of the CFG are assigned arbitrarily (e.g. using prime numbers or successive integers). These signatures are transferred to the WP explicitly by the checked processor. To do this, signature transfer statements are inserted before the compilation into the source of the checked program. The labeling is referred to as *encoding of the CFG*.

Derived run-time signatures This technique is used at assembly level. Nodes of the CFG represent branch-free instruction sequences (no jumps are allowed from or into the block), the signatures labeling the nodes are derived from the opcode sequence of the node by information compaction (mod-2 addition, a checksum or a linear feedback shift register, LFSR). The run-time signatures are derived by the WP concurrently, by compacting the fetch cycles on the instruction bus of the checked processor.

Stored reference The reference information (labeled CFG) is stored in the local memory of the WP in some form (e.g. list, adjacency matrix). If a run-time signature is received or computed by the WP then the stored reference is searched to find out whether the signature is a valid successor of the previous one. A general search and compare engine is implemented to handle the stored reference.

Embedded reference The reference information is stored in the checked processor and it is transferred to the WP in run-time, usually separated from the run-time signatures in some way. The hardware overhead is reduced significantly since no local storage is needed in the WP. However, the transfer of the reference information increases the time overhead of the checked processor.

Reference program The WP is a specialized processor which executes a reference program. The reference program has the same CFG structure as the program being executed by the checked processor. Each instruction of the reference program checks the actual run-time signature comparing it with the reference one and jumps to the next instruction (depending on the actual run-time information). The compiled graph tracing performed by the reference program is faster then the checking based on stored reference, and does not require a separate data structure. The architecture of the WP is simple, since only checking and control transfer instructions are implemented.

Two components of the correct control flow are checked by the WP methods:

- Checking the contents of the node, i.e. the type and sequencing of instructions within the node.
- Checking that the nodes are executed in an allowed sequence, i.e. that the main processor traverses the CFG correctly. It has to be pointed out that the checking verifies only that the nodes are executed in an *allowed* sequence, which is not necessarily the *correct* sequence of the program execution, due to data dependences which are not represented in the CFG. In case of conditional branches and selections, it is checked whether the next node belongs to the set of allowed successors, but the selection itself is unchecked (e.g. in an `if ...then ...else` statement, the successor belongs either to the `then` or to the `else` branch). Similarly, the number of cycles in repetition structures remains usually unchecked.

Assigned signature based methods check only the second component, since assigned signatures do not contain any information about the instructions belonging to nodes. Derived signature methods can check both. Table 4.4 gives a categorization of the typical methods discussed in the subsequent sections (Sections 4.5 and 4.6).

4.4.2 Memory access checking

Memory access checking by watchdog processors [NM82] is based on the capability-based addressing scheme [Fab74]. Code or data address ranges in the memory are represented by objects.

	Reference information		
	Stored reference	Reference program	Embedded reference
Derived sign.	ASIS, RMP, Checker	Cerberus-16, WDP	BPSA, GPSA, BAH
Assigned sign.	ESIC	SIC	SEIS

Table 4.4: Control flow checking methods

Each object has a capability (access right) for accessing another objects (e.g. read, write, execute). The set of objects and the corresponding capabilities are described in the local memory of the WP as an object table. The WP monitors the physical memory accesses of the checked processor and derives the active object (using the address of the fetched instruction), the accessed object (from the destination address of the instruction) and the actual operation (analyzing the instruction itself). The access right required to perform the current operation is compared to the enabled access rights stored in the object table. If there is no match, an access error is signaled.

As it turns out, the scheme is similar to the common protection mechanism provided by memory management units (MMU). Consequently, the technique has its grounds only in old-fashioned or (intentionally) simple systems without MMU.

4.4.3 Checking of control signals

The sequence of instructions executed by the processor is checked by control flow checking techniques introduced earlier. Similar techniques can be used to check the exact sequence of *control signals* corresponding to each opcode of a microprocessor. The scheme described in [Dan83] assigns a signature to each opcode by compacting the sequence of control signals using a multiple input LFSR. The reference signatures are stored in a ROM. When an opcode is fetched by the processor, it is also used to address the ROM. The reference signature is read and compared to the run-time one computed concurrently.

Microprogrammed control units can be monitored also by the methods proposed to check the instruction-level control flow. However, the special techniques used in these units (e.g. horizontal microinstructions) often result in inadmissible overhead and performance degradation. To overcome these problems, modifications aim at extending the microinstructions, inserting the necessary signatures in parallel [Nam82a], [ST82], [IK85]. Current research is focused on the synthesis of self-checking state machines and control units.

4.4.4 Reasonableness checks

Data errors can be detected by having the watchdog execute concurrent assertions. An assertion, as a notion of program correctness [Flo67], is an invariant relationship between the variables of a program. It is inserted by the programmer at different points in the program, signifying what he thinks (or intends) to be true for the variables. Assertions can be defined on the basis of the specification or some property of the algorithm (inverse of the problem, range of variables, relationships).

The use of assertions for detecting software and hardware faults is proposed in [And79] and [MMA84]. The effectiveness was demonstrated by measurements in [AB81]. Some designs proposed for executing assertions are sophisticated and complex, because they are aimed at not only detect errors but also recover the last verified state [LS84]. Special purpose RISC processors with hardware support for fast comparisons and range checking were proposed in [MM88].

The main objectives of the designs for concurrent execution of assertions are to keep the complexity of the WP as low as possible and to transfer the data from the main processor to the WP without significant overhead. There are two alternatives to solve these problems. In both schemes, the code of the assertions is stored in the local memory of the watchdog as a library of functions. Only identifier of the assertion function and the data have to be transferred to the WP.

If the WP checks applications in which the flow of data (the sequence of data values on the data bus) is known and invariant then the WP is able to capture the data by monitoring the data bus of the checked processor. Such special purpose watchdog processors [MML83] are proposed for telephone switching or flight control systems.

In general purpose systems the transfer of data can be organized by shared memory or a message queue. The first solution needs complicated synchronization techniques. The second one is discussed in [MEM85]. Before execution, the program is modified by replacing the assertion functions with a single statement which transfers the data values and the identifier of the checker function to the WP. The code of the assertion functions is downloaded into the local memory of the WP. In run-time, the WP receives the data and executes the required function. If the logical statement represented by the assertion function is false then an error is signaled.

4.5 Control flow checking using derived signatures

Derived signature based methods share the same principle: the nodes of the CFG are labeled by signatures that represent the instructions corresponding to the node. The WP computes the signatures in run-time concurrently and compares it to the reference provided earlier. In other words, the WP compacts the instructions executed by the checked processor, and periodically compares the intermediate results of the compaction, i.e. the signatures, to pre-computed references.

The first schemes were introduced in 1982 ([Nam82b] and [ST82]) followed by a number of improvements and modifications. In these methods, nodes of CFG are defined differently and the reference information is represented in various ways. Different processor architectures (traditional CISC, RISC with pipeline) and configurations (mono-, multiprocessors) are targeted. The authors try to find a compromise between the error coverage and the complexity (overhead) of the checking.

In the first method *Basic Path Signature Analysis* (BPSA, [Nam82b]) a node is defined as a branch-free sequence of assembly level instructions, the reference information is given by embedded signatures: the reference signature is inserted at the beginning of each node. The WP monitors the instruction bus of the processor and captures the reference signatures, using tag bits to differentiate them from the normal instructions. The main processor executes a NOP instruction whenever a signature is fetched. The WP computes the run-time signature concurrently. At the end of the node (signaled by a second tag bit) the run-time signature is compared to the reference one. Any discrepancy is taken as indication of an error.

The first modifications improved the following properties of BPSA (still using embedded reference signatures):

Complexity of the checking The hardware of the WP can be simplified if the reference signatures inserted at the beginning of a node are also included in the computation of the run-time signature. Additionally, they are defined in such a way that the run-time signature at the end of a node is all-one. A simple gate is enough to generate the error signal [ST82]. An other method [UR94] defines the nodes in such a way, that the run-time signature naturally becomes, or is forced by an embedded reference signature to become an m-out-of-n code.

The *Watchdog Assist* (WA+EPC, [SM89]) method defines the reference signature as the extended precision (check)sum of the instructions belonging to the instruction block of a node. The reference is transferred at the beginning of a block, and the opcodes of run-time instructions are subtracted. At the end of the block, the all-zero result is checked. The method provides the capability of a WT without requiring a separate WT unit: an error is detected if the result becomes negative before a zero check signal.

Overhead caused by the embedded reference signatures The insertion of the embedded reference signatures increases the memory overhead of the checked program and reduces the performance. A proper compromise has to be found between a required error latency (the number of instruction involved in a node) and the overhead.

The first method *Generalized Path Signature Analysis* (GPSA [Nam82b]) reduces the total number of signatures by checking paths rather than simple nodes. The CFG is divided into path sets, each set includes paths starting from the same node. Since there is only one reference signature for each path set, the run-time signature of each path in a set must be the same. It is assured by inserting *justifying signatures* in some paths. These signatures are involved in the computation of the run-time signature, so that the same signature results at the end of each path.

The second method *Branch Address Hashing* (BAH, [SS83]) can reduce the overhead for storing signatures by 50%. The nodes are defined differently and the reference signatures are inserted at the end of the node. In BPSA, each branch instruction should be preceded by a signature. In BAH, instead of embedding the signature, the branch address of the instruction is modified at assembly time in such a way that during execution, the changed branch address combined with the run-time signature results in the branch address taken actually by the processor. In the case of an error, the run-time signature and consequently the branch address are incorrect, resulting in a jump to an erroneous destination. The error is detected when the next reference signature is encountered and the actual run-time signature is checked. An explicit reference has to be inserted preceding each branch-in point. In error-free case, the run-time signature is all-zeros, making the comparison easier.

Increasing the error detection coverage The *Continuous Signature Monitoring with Instruction Hashing* (CSM+IH [WS88], [WS90b], [WS90a]) generalizes the idea of branch address hashing. Each instruction code is hashed in compile time by the compacted value of the previous instructions of the node, and re-hashed similarly in run-time by the run-time signature. The processor detects the invalid instructions and signals the error. This way, in some cases (processors with a large number of unused instruction codes) the coverage is increased and also the error latency can be reduced.

Asynchronous checking The previously described methods require the WP to be tightly synchronized with the main processor. An asynchronous version, the *Asynchronous Signatured Instruction Streams* is proposed in [ES84]. As the nodes of the CFG are executed by the main processor, the run-time signatures are computed by a hardware signature generator, the only part of the WP which is tightly synchronized with the processor. It contains a branch detection circuit to find out the boundaries of the nodes (an other possibility is to use tagged instructions in the main processor). The run-time signatures are transferred to the WP through a signature queue allowing an asynchronous checking. The peak rate requirements of the WP can be relaxed. This scheme can also be used to monitor more than one processor (it will be discussed later). The implementation [ST87] stores the reference signatures in the local memory of the WP.

The overhead caused by the embedded reference signatures can be completely eliminated if the reference information is moved to the environment of the WP. Moreover, it is possible to verify existing programs without re-compilation. Two categories are distinguished:

Stored reference information The *Roving Monitoring Processor* (RMP, [ST87]) based on the *Asynchronous Signatured Instruction Stream* method [ES84] described earlier implements a scheme based on stored reference signatures. The CFG is stored in a linked list format in the local memory of the WP. Receiving a signature from the signature queue, the WP examines whether it belongs to a node which can be reached from the previous (checked) one. The prototype of RMP is a specialized processor having only eight instruction types.

In the *Checker* of [MCS91], the reference signatures are stored in associative memory segments. The checking of the run-time signatures is fast (examining whether they are included in the reference signature database), but the complexity of the WP is high (see also in Section 4.7).

Reference program The first synchronous scheme which checks processors by a reference program is proposed in [Nam83]. The WP is implemented by a specialized processor called *Cerberus-16* which has a restricted instruction set. The program executed by the checked processor is mapped in compilation time to a program of the WP which has the same CFG structure. Each instruction of the WP includes the number of instructions in the corresponding node of the checked processor (to detect the end of a node), a justifying signature and finally the address of the next instruction in the WP. The reference signature of a path is computed by a data compression unit: at the end of a node, the justifying signature is fed to this unit. At the beginning of a checked path, the data compression unit is initialized, at the end of the path the reference signature is compared to the run-time one (both initiated by the corresponding WP instructions). If the checked processor executes a branch then it is signaled by the *branch detecting circuit* and the WP takes the next instruction given by the actual one (otherwise the successive instruction is executed).

A similar method, the *Watchdog Direct Processing* is described in [MLS91]. The program of the main processor can be checked without its modification. Control flow checking is achieved by direct address monitoring and a shadow watchdog program, instruction bit error checking is provided by an instruction compacting device. Due to the more detailed checking (branch address hashing, immediate address break detection etc.), the error coverage and latency is reported to be better than that of Cerberus-16.

In derived signature methods, run-time signatures can only be computed if the sequence of instructions executed by the checked processor can be captured by the WP. In the case of traditional processor architectures, it can be solved easily by monitoring the instruction bus of the checked processor. The sequence of instructions fetched by the processor is exactly the sequence executed by it. However, up-to-date processor architectures (e.g. Intel i486, Pentium [Cor94]), include on-chip instruction cache and prefetch queue. The sequence of instructions fetched by the processor can still be monitored by the WP but it is not exactly the sequence of instructions executed by the processor. Some instructions are executed in different order or are not executed at all.

The problem of monitoring pipelined RISC processors was solved in [DS91]. The main idea was that the reference signatures were computed on the basis of the instruction sequence *fetched* by the main processor, and the basic functions of the pipeline were imitated in the WP. The two events, which make a classical signature inconsistent are handled by some new notions. Delayed branching is resolved by *anticipated* signatures, always taking into account in the compaction (and in the computation of the reference) the delayed instructions following a branch, independently of the fact that they are executed or not. Exceptions and the corresponding flush of the pipeline are handled by *delayed* signatures, i.e. an instruction code is compacted only if a given number of successors are regularly fetched.

The other problem, the derived signature-based monitoring of processors with built-in cache remained unsolved. An external observer can not guess the instructions fetched by the processor from its cache, and the integration of the WP into the processor core can not be performed in off-the-shelf, commercial systems. The problem can be solved by assigned signatures methods, having the processor explicitly send the signatures to the WP. If the external signature transfer (i.e. not into the built-in cache) is not too slow compared to the operation of the cache then the performance degradation remains at an acceptable level.

4.6 Control flow checking using assigned signatures

Assigned signature control flow checking methods label the nodes of the CFG with signatures and check that the run-time sequence of signatures is allowed. Early methods checked the sequence by recording the execution and comparing it with sequences determined during programming or testing [KY75].

Structural Integrity Checking (SIC [Lu82]) was the first method which resulted in a much simpler implementation by making use of the control structure of the program. It introduced the typical software configuration used then in all assigned signature methods (Figure 4.2).

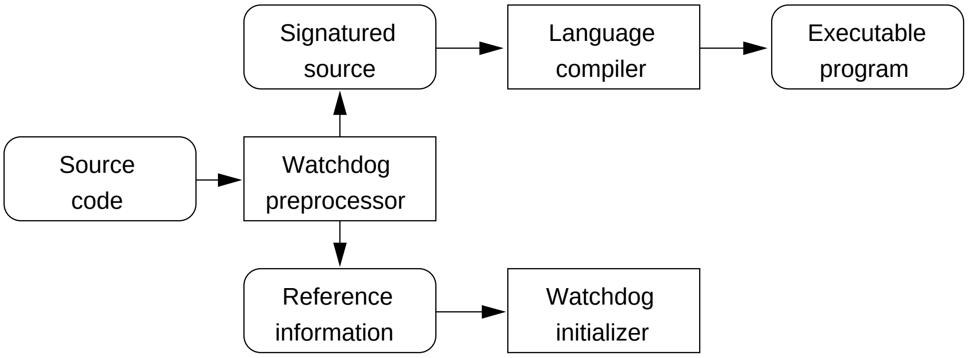

Figure 4.2: Software configuration of assigned signatures methods

The preprocessor reads the source of the checked program and assigns signatures to the nodes of the CFG. It works in four steps:

1. Construction of the CFG. The source is parsed, the control structures are identified and statements are associated with nodes of the CFG.

   Note that the preprocessor should be able to derive the CFG of the program using the source text, i.e. the control flow should be predictable (apart from the data dependent selections) during compilation. Some declarative programming languages (e.g. Prolog) do not have a program control flow which could be constructed on the basis of the program source, without executing the program. Programs written in these languages can not be checked by assigned signatures methods. Languages suited for structured programming (e.g. C, Pascal or Modula-2), which often used also in critical applications, meet the requirements.

2. Encoding of the CFG. Signatures are assigned to the nodes.

3. Modification of the program source. The program of the main processor is modified by inserting the statements which transfer the signatures to the WP.

   Note that signature transfer instructions should be inserted into the source text without modifying the original control structure (semantics) of the program, and resulting in a syntactically correct source. This requirement is easily met if the language contains the program construct known as concatenation, i.e. a single statement can be replaced by a block consisting of the concatenation of several statements. This way, a statement (which is associated with a node in the CFG) can be replaced by a concatenation of the signature transfer statement and itself. Structural languages can be modified in this way.

4. Extraction of the reference information for the WP.

Different methods represent the reference information and assign signatures to the nodes in different ways. These decisions determine also the checker structure.

In *Structural Integrity Checking* [Lu82], the nodes are encoded by selecting 8 bit integers randomly and independently, from a uniform distribution. The reference information is represented in the form of a reference program which has the same CFG as the checked program. In place of computations, it contains statements to receive and check the signatures from the main processor. The computational requirements of the WP are less than those of the main processor, however, since the reference program is extracted in the same language, the implementation requires a processor of the same type or a specialized compiler.

The ideal SIC method is defined in terms of two automata, one representing the checked program and a second one representing the execution of the reference program. The signature sequences of the checked program are described by a context-free grammar. The checker, as a push-down automaton, accepts the sequences described by the context-free grammar and rejects any others. In some cases, it is possible to use an equivalent regular grammar, which reduces the function of the checker to a finite deterministic automaton.

Extended Structural Integrity Checking (ESIC, [MH91]) extended the checking capabilities of SIC to check data-driven procedure calls and interrupt handlers. Nodes of the CFG are encoded by successive integers, the reference information is extracted in the form of a stored reference database, the valid successors of each signature are given in a sparse matrix format. The first and last nodes of a procedure are tagged by special flags: SOP (Start of Procedure) and EOP (End of Procedure).

The operation of the WP can be described as a finite deterministic stack automaton. States of the automaton are associated with nodes of the checked CFG, the next state is determined by a valid run-time signature. Invalid signatures are detected and the corresponding error is signaled. If a SOP signature is received (procedure call) then the actual state (pointer to the previous signature) is pushed to a WP-internal stack and the initial state corresponding to the called procedure is reached (i.e. the reference information of the called procedure is accessed).

After an EOP signature (return from a procedure call) the original state (pointer to the original reference signature) is popped from the stack.

Since SOP signatures are allowed at any point of the signature sequence, data-dependent procedure calls as well as interrupt handler procedures can be checked. It remains unchecked whether the correct procedure is called, but the return address is checked using the signature stack.

Signature Encoded Instruction Stream (SEIS) [PMHH93], [MPC+94] method was developed to check the control flow in state-of-the-art microprocessors with built-in instruction cache and instruction pipeline. It uses embedded reference information, thus the reference database and the search-and-compare engine are totally eliminated from the WP. The basic idea is, that reference signatures are transferred to the WP in parallel with the run-time ones, forming a new, compound signature. It means that this signature not only identifies the actual node of the CFG but also contains some information about the allowed successor nodes, i.e. successor signatures. The signatures of successor nodes are connected in a similar way like the tiles in the game of domino (Figure 4.3(a)), with the difference that the relationship between the connected parts of the signature (called labels) is defined by an f_n function (Figure 4.3(b)).

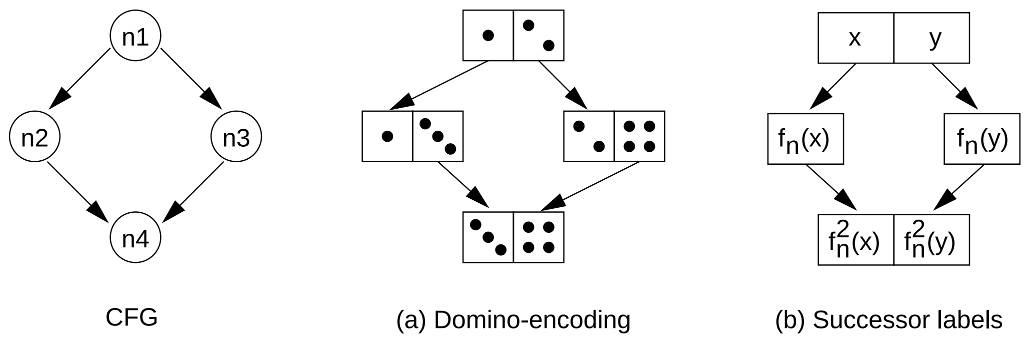

Figure 4.3: Basics of the SEIS encoding

As a consequence of the encoding outlined here, a signature is valid successor of a previous one if and only if at least one of its labels can be derived by applying the function f_n on one of the labels of the previous signature. If a path of the CFG is traversed by the program execution then it has to result in a sequence of signatures in which each signature (except the first one) is a valid successor of the previous one. If there is an edge which connects the node n_1 to node n_2 then the signature of n_2 should contain a label which can be derived using one of the labels of the signature corresponding to n_1.

The steps of the SEIS encoding algorithm are informally presented as follows:

1. The control flow graph of the program is extracted (Figure 4.4(a)). The basic control structures are assigned *elementary control subgraphs* which can be composed hierarchically.

2. The edges of the CFG are collected into an edge list (since the base of the encoding and checking is the existence of edges in the graph). The listing of the edges can be reduced to

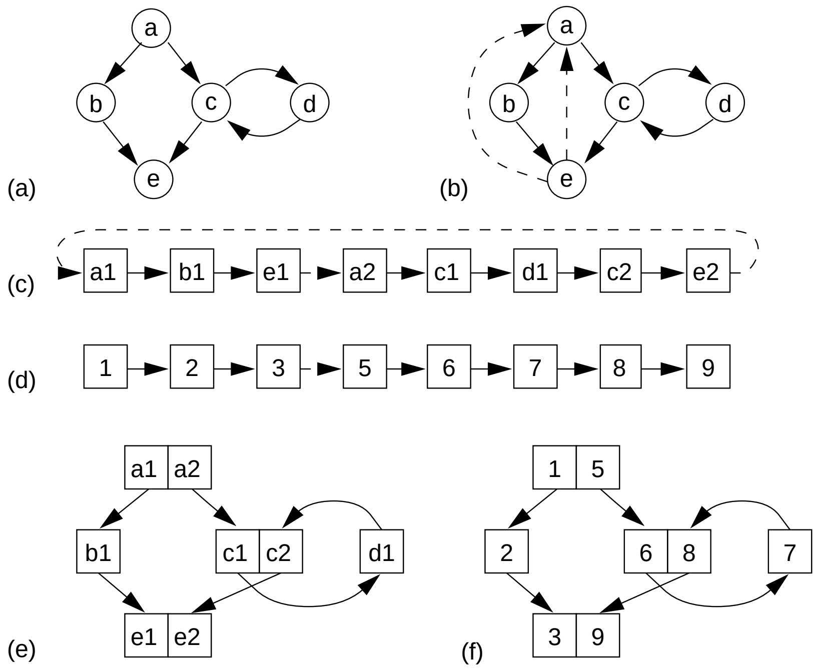

Figure 4.4: Encoding of the CFG in the SEIS method

the well-known problem of the *Eulerian circuit generation*. Inserting additional edges, the CFG can be transformed into an Euler-graph (Figure 4.4(b)) in which the Eulerian circuit can be generated using simple linear algorithms. This circuit contains all edges once and only once (Figure 4.4(c)).

3. The nodes (listed in the Eulerian circuit) are encoded. A cyclic ordering of the possible labels is defined by the f_n function. Subsequent nodes in the circuit which are connected by normal edges of the CFG are assigned subsequent labels; if they are connected by additional edges then they are separated by skipping a label (Figure 4.4(d)). In the Eulerian circuit, each node is listed several times (the number of occurrences of a node is equal to the maximum number of its input and output edges). The signature of a node is composed by concatenating the labels assigned to the multiple occurrences of it (Figure 4.4(e)).

4. In some nodes, the number of labels is reduced. The nodes corresponding to irregular control structures, which can have an unlimited number of successors (predecessors), are marked as *multiple output* (*multiple input*) nodes. (These nodes, their predecessors or successors can be identified on the basis of the syntax of the language.) The number of labels of the multiple input (multiple output) nodes have to be reduced. The basis of the reduction is, that signatures corresponding to multiple input (output) nodes may share the same label if this label has no successor (predecessor, respectively).

The SEIS graph extraction and encoding algorithm assures that if programs in C language are preprocessed then the number of labels of each node is limited by 3.

Procedure calls in the SEIS method are checked in a similar way like in the ESIC method. Synchronization of processes is checked by using *guard signatures*. They are transferred to the WP in the same way as the normal signatures, but their evaluation is different. Two type of guards are used: *initial guard*, inserted before a synchronization statement, and *checker guard*, inserted after it. The processes beginning a synchronization initialize the *communication registers* in the WP by sending the initial guards. A checker guard is evaluated in the WP by examining the required state of the communication registers (depending on the type of the synchronization).

The SEIS WP supports the rollback recovery of the checked system. When the main processor stores a checkpoint then simultaneously the state of the actual process is saved internally in the WP (initiated by a special command similar to the guard signatures). The WP-internal state of a process is its signature stack, so it has to be stored as checkpoint data. If a rollback recovery is performed then the stored stack space replaces the actual one, in this way the execution as well as the checking of the process can continue from the fault-free previous state.

The different levels of the software hierarchy are checked by autonomous modules of the WP hardware:

- The *statement level module* checks the actually received signature on the basis of the reference one. The possible label pairs are examined by 9 comparators and by a combinational circuit implementing the successor function. The time required to evaluate a signature is the set time of the comparators. After the evaluation, the register storing the reference signature is updated.

- The *procedure level module* checks the procedure calls by storing the reference signatures in the signature stack. The SOP and EOP signatures initiate a push or pop operation updating the stack pointer and the reference signature. The efficient utilization of the stack is assured by a dynamic splitting among processes (the cells are stored in a linked list format).

- In the *process level module*, the process ID is compared with the reference ID of the running process which was transferred by the scheduler. The guard signatures initialize and check the communication registers. The signature transfer is monitored by a timer which detects when a process fails to send signatures.

The advantages of assigned signature based methods are the relatively easy implementation of the preprocessor and the ability of asynchronous checking. The disadvantages can be summarized in four points:

- Performance degradation of the checked processor due to the explicit signature transfer. This problem originates in two subtask. The main processor has to generate the signatures (fetch, decode and execute the signature transfer instructions) and the signatures have to be transferred to the WP. Proposed solutions of these problems reduce the number of signatures (nodes of the CFG) or buffer them in the main processor serially and send them to the WP in parallel format [Lu82]. Both techniques introduce delays in the detection of errors.

- Inability to check existing programs without re-compilation. The preprocessor-based approach requires the availability of the source. The modification of the executable code arises problems, especially in highly optimized programs of RISC processors.

- Less fault coverage due to the fact that only the sequence of nodes is checked, the sequence of instructions represented by the node remains unchecked.

- The error latency can not be bounded easily. The nodes of the CFG are associated with high level statements which can represent an arbitrary number of assembly level instructions. The signature processing speed of the WP has to be fitted to peak rate requirements. Placing a queue between the processor and the WP introduces additional delays in the error detection.

An additional disadvantage of the mentioned first two methods is that the execution of the reference program and the search and compare engine in the stored reference needs a sophisticated WP implementation comparable in cost and complexity with the checked processor. The SEIS WP method eliminates this drawback completely.

4.7 Watchdog processors in parallel systems

Parallel system are commonly categorized as multi-computers or multiprocessors. Multiprocessors make use of a global shared memory and a conventional programming model. Multicomputers, further categorized as distributed systems, workstation clusters or massively parallel multi-computers, are scalable, consisting or loosely coupled, high performance computing nodes and clusters. In both main categories, concurrent error detection is a critical factor due to (i) the high number of components, (ii) the rapid spread of errors by cooperation between the components and (iii) the long computing time of the applications.

From the point of view of error detection, there are a few differences between multi-computers and multiprocessors [CP94]. In multi-computers, error detection can be provided by monoprocessor techniques which can be easily integrated into the computing nodes. Since error propagation is constrained to erroneous messages, it can be prevented by robust protocols and reasonableness checks. In multiprocessors, as state is shared by the processors and communication is not restricted to messages, the application of mono-processor techniques is often no longer satisfactory.

WP methods are traditionally developed for mono-processors, checking the computing core, which is the most intensively used resource and typically a core in checking the other system components. The extension of mono-processor solutions to multi-computers or multiprocessors requires the sharing of the WP among multiple nodes (processors), to keep the hardware overhead at an acceptable level. The number of nodes checked by a single WP is limited by the speed of the WP.

Several WP methods are proposed to check parallel systems by using shared watchdog processors [ST87], [MCS91], [MH91]. Since they do not check the interactions between the nodes, they are referred to as WPs in multiple processor systems.

The already introduced *Roving Monitoring Processor* (RMP, [ST87]) is based on the asynchronous signature checking method ASIS [ES84]. There is a hardware signature generator and a corresponding signature queue for each application processor. The RMP samples the queues and checks that the signature sequence corresponds with the stored reference CFG. The different processors are identified by a processor ID (used by the RMP to switch to the reference CFG of the actual processor). A prototype of RMP was able to monitor 16 processors (the speed of the checking was 300ns/signature).

A similar, shared WP is proposed in [MCS91]. The derived signatures are generated by hardware generators (including address break and interrupt detectors) attached to the application processors. The reference signatures are downloaded into the WP before the program run. Based on the address of the last instruction, a program segment identifier is derived for each signature, in such a way, that each segment contains a given (small constant) number of signatures. For each segment of the program there is a corresponding section of the WP memory where the allowed signatures are stored. A run-time signature is considered correct if it is present among the ones stored in this section. The verification is done by using fast associative search. The main advantages are that (i) the overhead of the reference memory is reduced since the structural information is not stored and (ii) the associative search is fast enough to serve several processors in a fashion similar to the above roving concept.

The assigned signature based ESIC WP [MH91] is implemented utilizing the advantageous properties of a transputer. Each application process is checked by an independent WP process. The signatures include process ID fields which control the fast context switch between these checker processes. The serial links of the transputer can be used to communicate between several WPs in a multiprocessor and to send reports and status information to a supervisory system. However, the possible co-operation of the WPs is not elaborated yet.

Due to its simple and fast hardware implementation, the SEIS WP is an optimal choice to be used as a shared WP in multiprocessors. The main problems of the previous methods were the following. First, the increase in the memory overhead was (over)proportional when more processes or processors were started. Second, the start of a new process always required the time-consuming download of the corresponding reference database. Additionally, the complexity of the control section of the checker was still considerably high. The SEIS WP eliminated these problems by using embedded reference information and a single reference signature to be stored in the WP. Moreover, it supports checking of scheduling as well as synchronization (communication) between processes.

Measure	Assigned signatures	Derived signatures
HW overhead	50–70% 1–2% (SEIS)	20% (embedded ref.) 60% (stored ref.)
SW overhead	15–20%	20% (embedded ref.) 0% (stored ref.)
Error coverage	60–70% (node sequencing)	> 99%
Error latency	30–40 instructions high variance	15–20 instructions low variance
Significant problems	Overhead	Cache in the checked processor

Table 4.5: Measures of control flow checking methods

4.8 Measures of control flow checking methods

Measurement results of overhead, error coverage, error latency of the mentioned various methods are presented in Table 4.5.

The analysis of Table 4.5 results in the overall conclusion that derived signature based methods have better measures than that of assigned signatures methods. However, their main drawback is that processors with built-in cache can not be checked at all. Assigned signatures methods have less error coverage, higher error latency (with high variance) and overhead, but they can check modern processor architectures.

4.9 Conclusion

In fault tolerant computing systems a single error detection mechanism can not assure a proper error detection. A combination of different techniques at circuit, system and application level has to be applied.

Watchdog processors are the only known methods which can increase the fault coverage of the traditional methods like those built into the MMU or a master-checker CPU configuration by detecting more than the half of the remaining errors. Such a combination can provide a good fault coverage at a price of a moderate hardware and time overhead. The impact of a high probability of detecting control flow errors is even much more than the fault coverage factor alone as a proper control flow is a prerequisite of the guaranteed activation of the built-in software error detection mechanisms like reasonableness checks.

Bibliography

[AB81] D. M. Andrews and J. P. Benson. An automated program testing methodology and its implementation. In *Proc. 5th Int. Conf. Software Eng.*, pages 254–261, 1981.

[AK84] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17:67–80, August 1984.

[And79] D. M. Andrews. Using executable assertions for testing and fault tolerance. In *Proc. 9th Int. Symposium on Fault Tolerant Computing*, pages 102–105, 1979.

[Avi81] A. Avizienis. Fault tolerance by means of external monitoring of computer systems. In *Proc. AFIPS Conference*, volume 50, pages 27–40, 1981.

[CC86] Y. Crouzet and J. Chavade. A 6800 coprocessor for error detection in microcomputers: The PAD. *Proc. IEEE*, 74:723–731, May 1986.

[CGH+93] M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hoenig, W. Hohl, E. Michel, and A. Pataricza. Fault tolerance in distributed shared memory multiprocessors. In A. Bode and M. Dal Cin, editors, *Parallel Computer Architectures - Theory, Hardware, Software, Applications*, volume 732 of *Lecture Notes in Computer Science*, pages 31–48. Springer Verlag, 1993.

[Cor94] Intel Corporation. *Pentium Processor User's Manual*, volume 3. Intel Co., S.l., 1994.

[CP94] M. Dal Cin and A. Pataricza. Increasing dependability in multiprocessors. In A. Pataricza, E. Selényi, and A. Somogyi, editors, *Proc. Symposium on Microcomputer and Microprocessor Applications uP'94*, volume I, pages 55–64. TU Budapest, 1994.

[CPW72] J. R. Connet, E. J. Pasternak, and B. D. Wagner. Software defenses in real time control systems. In *Proc. 2nd Int. Symposium on Fault Tolerant Computing*, pages 94–99, 1972.

[Daa86] A. Daam. The effectiveness of software error detection mechanisms in real-time operating systems. In *Proc. Int. Symposium on Fault Tolerant Computing*, pages 171–176, 1986.

[Dan83] S. F. Daniels. A concurrent test technique for standard microprocessors. In *Digest of Papers, COMPCON Spring 83*, pages 389–394, San Francisco, February 1983.

[DS91] X. Delord and G. Saucier. Formalizing signature analysis for control flow testing of pipelined RISC microprocessors. In *Proc. Int. Test Conference*, pages 936–945, 1991.

[ES84] J. B. Eifert and J. P. Shen. Processor monitoring using asynchronous signatured instruction streams. In *Proc. 14th Int. Symposium on Fault Tolerant Computing*, pages 394–399, 1984.

[Fab74] R. S. Fabry. Capability based addressing. *Communications of the ACM*, 17:403–412, July 1974.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32. American Mathematics Society, 1967.

[IK85] V. S. Iyengar and L. L. Kinley. Concurrent fault detection in microprogrammed control units. *IEEE Trans. on Comp.*, 34(9):810–821, September 1985.

[KGT89] J. Karlsson, U. Gunneflo, and J. Torin. The effect of heavy-ion induced single event upsets in the mc6809e microprocessor. In *Fault Tolerant Computing Systems*, volume 214 of *Informatik Fachberichte*, pages 296–307. Springer Verlag, Berlin, 1989.

[KY75] J. R. Kane and S. S. Yau. Concurrent software fault detection. *IEEE Trans. on Software Engineering*, 1(1):87–99, March 1975.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance - Principles and Practice*, volume 3 of *Dependable Computing and Fault Tolerant Systems.* Springer Verlag, 1990.

[Liu84] T. S. Liu. The role of a maintenance processor for a general purpose computer system. *IEEE Trans. on Comp.*, 33:507–517, June 1984.

[LS84] Y. H. Lee and K. G. Shin. Design and evaluation of a fault tolerant multiprocessor using hardware recovery blocks. *IEEE Trans. on Comp.*, 33:113–124, February 1984.

[Lu80] D. J. Lu. Watchdog processors and VLSI. In *Proc. Nat. Electron. Conf*, volume 34, pages 240–245, Chicago, October 1980.

[Lu82] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Trans. on Comp.*, 31(7):681–685, 1982.

[MCS91] H. Madeira, J. Camoes, and J. G. Silva. A watchdog processor for concurrent error detection in multiple processor systems. *Microprocessors and Microsystems*, 15(3):123–131, April 1991.

[MEM85] A. Mahmood, A. Ersoz, and E. J. McCluskey. Concurrent system-level error detection using a watchdog processor. In *Proc. 15th Int. Symposium on Fault Tolerant Computing*, pages 145–152, 1985.

[MH91] E. Michel and W. Hohl. Concurrent error detection using watchdog processors in the multiprocessor system MEMSY. In *Fault Tolerant Computing Systems*, volume 283 of *Informatik Fachberichte*, pages 54–64. Springer Verlag, 1991.

[MLS91] T. Michel, R. Leveugle, and G. Saucier. A new approach to control flow checking without program modification. In *Proc. 21st Int. Symposium on Fault Tolerant Computing*, pages 334–341, 1991.

[MM88] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Trans. on Comp.*, 37(2):160–174, 1988.

[MMA84] A. Mahmood, E. J. McCluskey, and D. M. Andrews. Writing executable assertions to test flight software. In *Conf. Rec. 18th Annual Asilomar Conf. Circuits and System Conf.*, pages 262–266, 1984.

[MML83] A. Mahmood, E. J. McCluskey, and D. J. Lu. Concurrent fault detection using a watchdog processor and assertions. In *Proc. Int. Test Conference*, pages 622–628, 1983.

[MPC+94] I. Majzik, A. Pataricza, M. Dal 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 *Lecture Notes in Computer Science*, pages 386–403. Springer Verlag, Berlin Heidelberg, 1994.

[MQS90] H. Madeira, G. Quadros, and J. G. Silva. Experimental evaluation of a set of simple error detection mechanisms. *Microprocessing and Microprogramming*, 30:513–520, 1990.

[Nam82a] M. Namjoo. Design of concurrently testable microprogrammed control units. In *Proc. 15th Workshop on Microprogramming*, pages 173–180, 1982.

[Nam82b] N. Namjoo. Techniques for concurrent testing of VLSI processor operation. In *Proc. Int. Test Conference*, pages 461–468, 1982.

[Nam83] N. Namjoo. Cerberus-16: An architecture of a general purpose watchdog processor. In *Proc. 13rd Int. Symposium on Fault Tolerant Computing*, pages 216–219, 1983.

[NM82] M. Namjoo and E. J. McCluskey. Watchdog processors and capability checking. In *Proc. 12nd Int. Symposium on Fault Tolerant Computing*, pages 245–248, 1982.

[OCKB75] S. M. Ornstein, W. R. Crowther, M. F. Kraley, and R. D. Bressler. Pluribus - a reliable multiprocessor. In *Proc. AFIPS Conf.*, volume 44, pages 551–559, Anaheim, Canada, 1975.

[ORG92] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In *Proc. 22nd Int. Symposium on Fault Tolerant Computing*, pages 316–325, 1992.

[PMHH93] A. Pataricza, I. Majzik, W. Hohl, and J. Hoenig. Watchdog processors in parallel systems. *Microprocessing and Microprogramming*, 39:69–74, 1993.

[SL81] D. P. Siewiorek and L. K. Lai. Testing of digital systems. *Proceedings of the IEEE*, 69:1321–1333, October 1981.

[SM89] N. R. Saxena and E. J. McCluskey. Control flow checking using watchdog-assists and extended precision checksums. In *Proc. 19th Int. Symposium on Fault Tolerant Computing*, pages 428–435, 1989.

[Sos94] J. Sosnowski. Transient fault tolerance in digital systems. *IEEE Micro*, pages 24–35, February 1994.

[SS82] D. P. Siewiorek and R. S. Schwarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[SS83] J. P. Shen and M. A. Schuette. On-line self-monitoring using signatured instruction streams. In *Proc. Int. Test Conference*, pages 275–282, 1983.

[ST82] T. Sridhar and S. M. Thatte. Concurrent checking of program flow in VLSI processors. In *Proc. Int. Test Conference*, pages 191–199, 1982.

[ST87] J. P. Shen and S. P. Tomas. A roving monitoring processor for detection of control flow errors in multiple processor systems. *Microprocessing and Microprogramming*, 20:249–269, 1987.

[STDM82] M. E. Schmid, R. L. Trapp, A. E. Davidoff, and G. M. Mason. Upset exposure by means of abstraction verification. In *Proc. 12th Int. Symposium on Fault Tolerant Computing*, pages 237–244, 1982.

[TA80] S. M. Thatte and J. A. Abraham. Test generation for microprocessors. *IEEE. Trans. Comp.*, 29:429–441, June 1980.

[TDM+94] K. Tilly, T. Dobrowiecki, I. Majzik, A. Somogyi, A. Várkonyi-Kóczi, I. Kiss, Z. Dankó, Cs. Tóth, B. Vadász, and T. Zsemlye. The application of intelligent diagnostic centers in the implementation of distributed fault-tolerant systems. Technical Report TUB-TR-94-EE09, Technical University of Budapest, Budapest, 1994.

[UR94] S. Upadhyaya and B. Ramamurthy. Concurrent process monitoring with no reference signatures. *IEEE Transactions on Computers*, 43(4):475–480, April 1994.

[Wak82] J. Wakerly. *Error Detecting Codes, Self Checking Circuits and Applications*. Elsevier Science Publishing Co., Inc., Amsterdam, New York, 1982.

[WS88] K. Wilken and J. P. Shen. Continuous signature monitoring: Efficient concurrent detection of processor control errors. In *Proc. Int. Test Conference*, pages 914–925, 1988.

[WS90a] K. Wilken and J. P. Shen. Concurrent error detection sing signature monitoring and encryption. In *Proc. Int. Working Conference on Dependable Computers in Critical Applications*, New York, 1990. Springer Verlag.

[WS90b] K. Wilken and J. P. Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer-Aided Design*, 9(6):629–641, June 1990.