Accepted to the International Journal of Computer Systems Science & Engineering

Multiprocessor Checking Using Watchdog Processors

I. Majzik[++], W. Hohl[+], A. Pataricza[+,++], V. Sieh[+]

[+] Universität Erlangen-Nürnberg, IMMD III, Germany
[++] Technical University of Budapest, BME MMT, Hungary

hohl@informatik.uni-erlangen.de

Abstract. A new control flow checking scheme is presented, based on assigned-signature checking using a watchdog processor. This scheme is suitable for a multitasking, multiprocessor environment. The hardware overhead is comparatively low because of three reasons: first, hierarchically structured, the scheme uses only a single watchdog processor to monitor multiple processes on multiple processors. Second, as an assigned-signature scheme, it does not require monitoring the instruction bus of the processors. Third, the run-time and reference signatures are embedded into the checked program; thus, in the watchdog processor neither a reference database nor a time-consuming search and compare engine is required.

1 Introduction

Massively parallel computing systems running computing intensive applications demand a high degree of fault-tolerance. Fault-tolerance techniques require error detection mechanisms with high coverage and low latency. As the majority of failures results from transient faults, concurrent fault detection is of utmost interest. However, with increasing number of processing units and parallel processes, concurrent fault detection becomes more and more difficult.

Since the majority of transient processor faults results in control-flow disturbances, a widely used concurrent error detection method is concurrent control flow checking using a *watchdog processor* (WP). A WP as a relatively simple coprocessor compares the actual control flow - represented by run-time *signatures* - with the previously computed reference control flow. This approach offers the possibility to connect a single WP to multiple processors, reducing the hardware overhead.

Most of the WP implementations presented in the literature check single processors [10]. They can be classified according to the way how run-time signatures are generated and reference signatures are stored. Some typical methods are presented in Table 1. The methods using *derived* run-time signatures monitor the state of the processor bus. *Assigned* run-time signatures are computed and inserted into the program source by a precompiler; they are transferred to the WP by the checked processor itself. The reference is either a *stored database* of the admissible signature sequences or a special *WP program* of signature evaluation instructions [7]. (In [18] the main processor itself emulates the signature checker by utilizing unused resources). A further possibility is to transfer the reference signatures to the WP at run-time explicitly, using special instructions *embedded into the program* of the checked processor.

There are also two approaches to integrate watchdog processors into multiprocessor systems. The Roving Monitoring Processor [19] is connected to multiple processors and monitors their states sequentially without checking their interactions. The Checker described in [9] stores the reference signatures in the local memory of the WP. The information on the control flow graph (CFG) is not stored, the admissible run-time signatures are identified by associative memory segments in the WP. Multiple processors are checked using signature queues. A further WP to be used in multiprocessors is the Extended Structural Integrity Checker (ESIC [14]). Signatures are assigned based on the high-level language structure of the program and transferred to the WP explicitly. Reference signatures are downloaded to the WP in tabular form before the beginning of program execution. The WP receives the run-time signatures and works as a finite deterministic stack automaton. In a multitasking environment, the WP always switches to the reference table of the process a signature was received from.

		Reference		
		Stored signature database	Watchdog program	Embedded signatures
Run-time control flow	Derived signatures	Asynchronous Signatured Instruction Stream [4]	Watchdog Direct Processing [13]	Basic Path Signature Analysis [21]
	Assigned signatures	Extended Structural Integrity Checking [14]	Structural Integrity Checking [8]	Signature Encoded Instruction Stream [17]

Table 1 Control flow checking methods

The main drawback of these approaches is the (over)proportional increase of hardware and time overhead if more computing nodes and processes are added. Our paper presents a novel program control-flow checking method and a corresponding WP architecture called *Signature Encoded Instruction Stream* (SEIS [17]). The design goals of the SEIS project were:

- Efficient *checking* method of *multiple processors* using a single WP
- Checking *interactions between the processes* of an application
- Reducing the hardware overhead by efficient utilization of the WP resources.

As up-to-date microprocessors have a built-in instruction pipeline and on-chip cache memory, the assigned signature method was chosen as the focus of interest. The experimental multiprocessor system MEMSY (Modular Expandable Multiprocessor System) [1],[2],[3] was used as test-bed for the SEIS WP prototype.

The paper is structured as follows. The next section presents the checking schemes applied on different levels of the target system covering both theoretical and hardware aspects. Section 3 describes the integration of the SEIS WP into MEMSY and Section 4 presents measurement results of the prototype implementation.

2 Levels of Concurrent Error Detection

Our method is intended for use in multiprocessors with a UNIX-like operating system, widely used in massively parallel multiprocessors for scientific computations. An application consists of processes running the application program written in a procedural programming language. At each abstraction level: process, procedure, and statement, different checking methods and WP modules are used.The following run-time informations are monitored and therefore, included into a signature:

- the *statement label* consisting of three sublabels identifying the location and the valid successor statements in the procedure
- the *procedure ID* identifying the procedure of a process
- the *process ID* identifying the application process
- the *ID of the processor* which has sent the signature
- *synchronization labels* (special guard signatures).

The lower level checks are independent from the upper level ones; each level forms a self-contained, independent module. Each of the checks on the different levels can be executed simultaneously, assuring high operating speed. The checking modules are summarized in Table 2. An error is detected if any one of the checker modules reports an error.

The checker hierarchy can detect the majority of faults at the level of their first manifestation. A fault in the program counter results in an invalid sequence of statements; it can be detected either as a wrong statement label or as a signature time-out. Stack pointer faults can result in a faulty procedure return detected by the procedure level checker. Permanent software or transient hardware faults during synchronization are detected by the process level synchronization checkers.

Checker level	Checked operation	Signature information	Checker method
Statement	Statement sequence	Statement labels	Comparison
Procedure	Call and return	Procedure ID	Reference stack, comparison
Process	Scheduling	Process and processor ID	Processor-process database check
	Synchronization	Guard signatures	Reference label generation and comparison
	Signature transfer rate	Signatures	Basic timer

Table 2 Hierarchical checking (summary)

2.1 Statement Level Checking

The execution sequence of statements in a program can be associated with a *program control flow graph (CFG)*. *Vertices* represent branch-free statement sequences, *edges* represent the syntactically correct control flow between them. The CFG can be extracted by syntax analysis of the program source. Interrupts, data dependencies in conditional branches, and procedure calls referenced by pointers raise special problems. Conditional branches allow typically two outgoing edges from a vertex, procedure calls may call any other procedure, and interrupts, resulting in a call to an interrupt handling procedure, may occur at any time. The latter two problems belong to the procedure level and are covered in the next subsection.

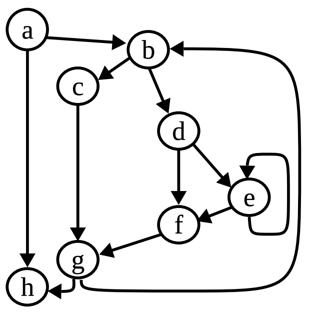
The *statement level WP module* checks the correct execution order of statements by comparison with the corresponding paths in the CFG. In order to identify the state of program execution, statement labels are assigned to the vertices of the CFG. These labels are explicitly transferred to the WP. The transfer instructions and the label values are inserted into the high level source text by a precompiler.

Statement labels identify not only the CFG vertices but their *(syntactically) valid successor vertices* as well. Thus, checking of the statement label sequence is based only on the presently checked label and its predecessor. This eliminates the need of a WP reference database. Hence, the evaluation of the correctness of program flow is a simple combinatorial task without any time consuming database search, allowing high speed processing. The label assignment algorithm of the precompiler is as follows (for the formal description see [12]):

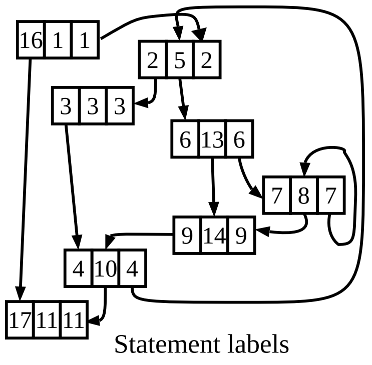
1. The *CFG* of the procedure is *extracted*. The basic control structures form subgraphs of the CFG. These subgraphs are identified according to the requirements of the encoding algorithm: that is, the number of successors of a vertex is limited in order to reduce the information to be encoded in the label identifying them. The subgraphs are composed to form the CFG of a procedure.
2. The edges of the CFG are collected into an *edge trail*. The problem of edge collection can be solved by well-known methods of Eulerian circuit generation.
3. A *cyclic ordering of label values* is defined and the edge trail is *encoded*. Adjacent vertices of the CFG are encoded by subsequent label values and different trails are separated by unused sublabels. After encoding the trail, all labels corresponding to the same vertex (called *sublabels*) are concatenated defining the *statement label*. In this way *a statement label is a valid successor of a reference label if and only if one of its sublabels is successor of one of the sublabels of the reference label*. This is the basic rule of the statement label checking.
   Fig. 1 presents an example C program, its CFG and the corresponding sublabel set. Using the simplest, natural ordering of the sublabels, a sublabel *j* is a valid successor of a reference sublabel *i* if and only if *j=i+1*. This rule is implemented by the successor function *F* which increases the reference sublabel value by one. The statement label sequence during the execution of the program is valid if the subsequent statement labels have successor sublabels. In the example vertex *d* is a valid successor of vertex *b*, since *F(2,5,2)=(3,6,3)* and *(6,13,6)* have 6 as common sublabel.
4. *Intermittent signatures* are used in the encoding of special control structures with a large number (>3) of successor or predecessor vertices. The number of such intermittent signatures (and the time overhead resulting from multiple signature transfers in a single vertex) is limited in a single signature per vertex by using a slightly modified encoding algorithm. This is based on the reuse of identical sublabels in different vertices without introducing ambiguity in the encoding [12].

```
a:for (j=0;j<2;j++){
b: if (x<0){
c:   x=x+8;
   }else{
d:   while (i<3){
e:     i=x+i;
f:   }
g: }
h:}
```

Program text



Program control-flow graph



Statement labels

Statement label set:

Vertex	Sublabels	Vertex	Sublabels
a	(16,1,1)	e	(7,8,7)
b	(2,5,2)	f	(9,14,9)
c	(3,3,3)	g	(4,10,4)
d	(6,13,6)	h	(17,11,11)

Fig. 1 Encoding of the program control-flow graph

Assume that a `switch` statement with an actual sublabel of *s* has more output branches than *3*, the maximal number of successor vertices allowed by the basic encoding scheme. The sublabel *F(s)* is assigned to each successor vertex, indicating that they are all valid successors of the `switch` vertex (*F* is the successor function on the sublabels). Note, that no data dependencies, like branch selection, are checked by the WP. The individual output branches of the `switch` statement are distinguished by assigning different second and third sublabels to the vertices.

In order to keep the memory and time overhead at an acceptable level, the number of statement labels in a procedure can be reduced. This reduction is performed on the CFG before the encoding step. It can be either static or dynamic.

Static reduction decreases the number of vertices in the CFG and thus the signature transfer instructions in the program code by merging multiple statements into a single vertex and correspondingly into a single signature. A user-defined *static reduction factor* controls the number of statements merged. Higher numbers increase error latency and reduce the probability of error detection, yet on the other hand result in fewer checks and shorter execution time. Static reduction may remove small branches in the CFG. Increasing the *statement factor* without enabling the static reduction merges only branch-free statement sets into a single vertex.

Removal of cycles in the CFG is not allowed, because otherwise the program may run within loops for extended periods of time without any check. Hence, each loop has to contain at least one statement label. Overhead measurements (described in Section 4) have shown a very high bus traffic due to short loops inducing burst-like transfers of many signatures. *Dynamic reduction* has proved efficient to avoid this effect. Instead of transferring a signature, only a counter variable is incremented. If the signature counter exceeds the user-defined *dynamic reduction factor,* the counter is reset and a signature is transferred to the WP. A similar reduction can be achieved for a predefined reduction factor by *loop unrolling* followed by a static reduction phase.

The hardware implementation of the *statement-level checker* is quite simple, due to the efficient CFG encoding (Fig. 2). Only the reference statement labels have to be stored and regularly updated. The successor function of the sublabels can be implemented as a combinational logic circuit, the evaluation of the statement sublabels is performed by a comparator set.

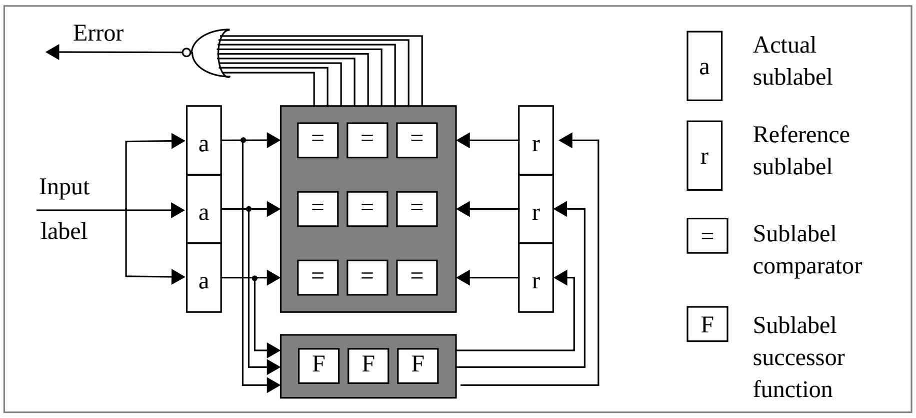


Fig. 2 Statement level checker module

2.2 Procedure Level Checking

The *procedure level checker module* has to check the procedure calls and returns. Upon a procedure call, the WP has to push its state, represented by the previous signature, onto its stack (called the *signature stack*); upon return, the latest signature has to be popped from the stack.

Procedure calls are potentially data-dependent (e.g. procedure calls through variables) in high level languages. Neither the location of the procedure call nor the called procedure can be identified by the precompiler in the CFG extraction step. Hence, procedure calls are allowed at any location of the program, independently of the actual instruction structure. Function calls embedded into arithmetic expressions and interrupt handler routines can be checked in the same way as procedure calls. The disadvantage is that only the returns from procedures can be checked, i.e. a wrong procedure call will be detected only after a long latency. Nevertheless, procedure calls are allowed to start only at an entry point of a procedure, so an erroneous jump to the starting point of a procedure can not be detected immediately.

The first and last statement labels of procedures are marked by flags: *Start of Procedure* (SOP) and *End of Procedure* (EOP), respectively. SOP means that the WP has to push the actual reference onto the stack and the actual statement label is valid as the first reference of the called procedure. In case of EOP the statement label has to be validated by the statement label checker and the next reference has to be popped from the stack (the reference of the calling procedure).

The procedures of a program are numbered and their identifiers are embedded into the signatures, together with the statement labels. *The procedure ID*s are allowed to change only if the SOP flag is set.

In a multi-tasking environment the WP and the signature stack storage is shared between different processes. Signature stack areas can be either statically or dynamically allocated. Static allocation is uneconomical if there are “hyperactive” (e.g. recursive) processes needing more stack space, while others hardly use the stack. In the case of the dynamical allocation strategy the individual stacks are parts of a single global stack area implemented as a linked list. Each process stack is defined by a pointer as header of a linked list. Cells of a stack can be linked to and from a global free list consisting of the whole unused area. Thus, the stack area of a single process is limited only by the global number of free cells and the activity of the other processes.

The procedure-level hardware checker module consists of the *comparator for procedure IDs* and the *stack maintenance control*. The size of the signature stack storage depends on the number of admissible embedded procedure calls. In the case of a *stack overflow* all active processes have to be stopped and the stack content is stored into its virtual extension in the main memory, or in a stable storage, from where the stack can be reloaded after becoming empty.

2.3 Process Level Checking

The process level module checks the *scheduling of application processes* running on the same processor and the *interaction of different processes*, i.e. synchronization. Signature transfer times are monitored by a timer and can be used to detect a hung process.

Checking of Process Scheduling. A unique ID is assigned to each application process. A *processor-process database* is established in the process level checker module of the WP: each processor has a record in this database storing the ID of the presently running process. If the operating system schedules a new process on a processor then the corresponding record is replaced by the new process ID. The process and processor IDs are embedded into the signature; thus, the WP can compare them with the record in the processor-process database, allowing only correctly scheduled processes to be active.

Signature transfers are monitored process-wise by separate logical *time-out checkers* in the WP activated by the scheduler. All time-out checks share the same physical timer of the WP.

Checking of Process Interaction. Faulty process interaction and synchronization based on sensitive data structures like semaphores (memory-based synchronization) or message descriptors/headers (message-based synchronization) may result in the propagation of errors from faulty processes to the other ones. To prevent these effects, the process interactions need to be carefully checked as well.

The checking of complex communication and synchronization mechanisms requires the formal description of the fault-free operation, the possible faulty operation and the error detection technique. It provides not only a clear description but also makes the formal reasoning on the efficiency of the error detection mechanism possible (e.g. error coverage of the technique, effects of the faulty operation).

The basic idea is illustrated by the simplest example, the synchronous communication between two processes. If the sender process terminates correctly after the receiver process has accepted the data and the receiver cannot get data before the sender begins the communication then the transfer may be considered valid. An error is detected if one of the partner processes has already finished the communication and the other partner has not even started it.

The approach of the error detection is similar to the lower levels checks. The WP is notified on the status of the interprocess control flow by special signatures. Such a signature should change only after a synchronization in order to distinguish the different phases of the individual processes during the cooperation. In this way a local checking is performed, i.e. only the synchronization statements are guarded by the special signatures.

Based on the reference labels, the checking is performed by the WP in two phases:

1. The first, *initializing signature* before the execution of the synchronization notifies the WP, with which partner process the synchronization is intended. It contains the ID of the presently running process and the ID of the intended partner for communication. In case of error-free operation, the other process will send a similar signature referring to the first process prior of the synchronization.
   Based on the initialization signatures of the participating processes the WP internally generates a common reference signature for both of them: A *reference register* is reserved for each process internally in the WP. When receiving this type of signature, the WP checker module examines the reference register of the partner process. If the reference register does not contain the ID of the running process, then the process is the one beginning the synchronization and the partner is not initialized. To indicate this, the checker module stores the ID of the partner in the reference register of the running process.
   However, if the reference register of the partner already contains the ID of the running process, the partner is ready for the synchronization due to the processing of a previous signature. The initialization can be finished. The WP stores in both reference registers the same reference label identifying the pair of partner processes (e.g. by some function of the process ID bits). This indicates that both partners are ready for the communication.
2. The second, the *reference signature* is sent during the synchronization itself by each participating process. It is computed by the precompiler at compilation time in the same way as the WP prepares internally the reference label in the initialization phase. The WP checker module compares the reference signature with the reference label. If there is a mismatch then an error is detected. (The reference signature is valid only if the partner processes coincide with the expected ones, and both processes have already initialized the synchronization.) Additionally, a time-out is generated if the reference signature does not arrive in a given time interval after the initialization signature.

The formal description used in this case is a simple CCS-like process algebra [15] due to its inherent property of expressing synchronous communication. Table 3 presents the compact description of the original communicating processes P and Q, the insertion of initialization and reference signatures and the model of the checker (the time-out checking is not included). The analysis of the reachability tree of the checked process system shows that in case of the presented faulty application (process Q misses the message and the synchronization) either the *Error* action is observable (if the faulty process Q_{sf} is scheduled first) or the time-out signal is generated (if the process P_s is scheduled first).

	Process terms

Original process system	Original_application = P | Q
	P = message.P’ |
	Q = message.Q’

Modified (signatured) process system	Signatured_application = P_s | Q_s = init(p,q).message.checker(p⊗q).P’ | init(q,p).message.checker(p⊗q).Q’
A faulty, signatured process system	Faulty_application = P_s | Q_{sf} = init(p,q).message.checker(p⊗q).P’ | init(q,p).checker(p⊗q).Q’
Checker process	Checker = init(p,q).(init(q,p).checker(p⊗q).checker(p⊗q).0 + checker(p⊗q).Error) + init(q,p).(init(p,q).checker(p⊗q).checker(p⊗q).0 + checker(p⊗q).Error)
Checked faulty process system	Faulty_application | Checker

P’, Q’ processes represent the actions following the message transfer,
init(p,q) and *init(q,p)* represent the transfer of different initialization signatures,
checker(p⊗q) represent the transfer of the common reference signature,
Error action represent the error signal.

Table 3 Checking of the synchronization

2.4 Additional Features of the Watchdog Processor

The SEIS WP is designed to support *rollback recovery* in a massively parallel multiprocessor. The checked system regularly stores the states of the processes in a stable storage. In case of an error the application is restarted from the saved state avoiding the loss of the whole computing time.

Checkpointing and restarting the system requires the WP to save and restore the signature stacks of all processes affected in the main processor. Checkpoints are stored as dynamically linked lists in the global stack space. The implementation of the checkpoint operations increases only the complexity of the stack maintenance hardware, other WP modules were not changed. Checkpoints may share stack cells with the actively used reference stack, thus time and space consuming stack copying is avoided. Checkpointing only requires saving the operational stack pointer, and write-protecting the reference stack. After an EOP signature labelling a return statement from a subroutine, a write-protected stack cell is not linked to the free list, but remains part of the checkpoint space. Thus, the internal checkpoint operations of the WP can be executed in a predefined time independent from the stack depth of the process. The following operations are supported:

- *Generation of a tentative checkpoint*: The previous tentative checkpoint in the WP is replaced by the actual reference signature stack of the process
- *Commitment*: The tentative checkpoint in the WP is made permanent
- *Roll-back recovery*: The operational reference signature stack of the process is replaced by the permanent checkpoint.

The WP executes these operations internally initiated by corresponding special commands embedded in the signature flow.
If an error is detected by a checker module of the WP, an error status word is generated and the checked system is alarmed by an interrupt. The error status word is the concatenation of the results of the different checker modules.

3 An Experimental SEIS WP for the MEMSY Multiprocessor

The MEMSY multiprocessor developed at the University of Erlangen-Nürnberg has a 2-level hierarchical, scalable regular structure with distributed locally shared communication memory [2], [3]. The processing nodes at each level form a four-neighbor toroidal mesh coupled by multiport memories. Locally shared memory modules allow communication of two neighboring nodes with the help of an interrupt network. This communication memory is mapped into the address range of the processors and interfaced through dedicated buses. The basic building block of the MEMSY architecture is an elementary pyramid consisting of one higher level node supervising four lower level nodes. Each computing node is a multiprocessor itself, containing four MC88100 RISC processors with the corresponding cache and MMU chips. The processor modules are off-the-shelf, highly integrated boards; so the instruction bus of the processors is not observable for the purposes of derived signature generation without drastic hardware modifications.

Each basic pyramid of MEMSY is checked by a single WP in order to reduce the hardware overhead. Thus, the WP is able to simultaneously check 5 computing nodes consisting of a total of 20 processors and running a maximum of 1280 processes (max. 256 processes per node) [11].

The WP as a multiport coprocessor is connected to the five computing nodes in an elementary pyramid. The requests on the input ports of the WP are served sequentially according to a round-robin priority scheme. A signature is transferred within a single communication memory cycle. An input FIFO is used to smooth out the time overhead of the relatively complicated checkpoint operations and to avoid delays due to the time-shared use of the WP. Control operations, like initialization etc., are performed by the higher level main processor node in an elementary MEMSY pyramid. All error reports generated in the WP are copied to the higher level node, forming an error log of the entire basic pyramid.

The WP is implemented as a 16 MHz coprocessor board on the VME bus of the higher level node with interfaces to the four computing nodes on the lower level identical with those used for the communication memory. WP operations (arbitration, signature evaluation, stack handling and checkpointing) are controlled by 6 MACH230 PLDs (3600 gate equivalent per device). The signature stack is in a 256 K * 64 bit RAM block which is oversized if no recursive programs are running. Synchronization checks were not used in the experimental version of the WP. Worst case signature transfer and evaluation time is even in this moderate speed experimental version as low as 300-600 ns depending on the signature type and number of simultaneous requests. Tentative checkpoint generation is executed in 2.3 microseconds.

The SEIS C precompiler was implemented in C (fully portable code of about 4000 lines). Since the WP is memory-mapped, the signature transfer statements are simple instructions inserted into the high level C source. The signature data word contains the statement label, while the statement identification (processor, process and procedure IDs) is transferred in the address word undergoing a logical to physical address translation in the MMU. During process initialization, the MMU is programmed in such a way that the uniform logical processor and process IDs (assigned in the precompilation phase) correspond exactly to the IDs assigned actually by the operating system. Thus, code sharing between processes is allowed.

4 Measurement Results

Standard benchmarks (like *whetstone, dhrystone, linpack* etc.) and scientific calculation programs representing the expected typical MEMSY user profile (like a *multigrid based solver of Poisson differential equations*) were used for verification. The *SEIS C precompiler* inserted the signature transfer statements, compilation was done using *gcc version 2.2.2* generating a highly optimized code. The following characteristics were measured for each reduction strategy:

- *run-time overhead* (number of signatures sent to the WP)
- *distribution of the number of steps between subsequent signature transfers* in terms of number of instructions executed
- *overhead in static code length*
- *fault coverage*.

Naturally, the resulting characteristics show a dependency on the benchmark and on the reduction method applied. However, the basic trends were essentially identical. Therefore, the following analysis of the measurement results is illustrated mainly by the detailed results of the multigrid solver. The experiences with other benchmarks will be presented as accumulated intervals.

4.1 Run-time overhead

The run-time overhead is proportional with the number of signatures sent to the WP. This parameter can be controlled by the static and dynamic reduction factors. The reduction of signatures in a C program fragment is demonstrated in Table 4. The static reduction factor defines the maximum number of statements which are associated with a single signature, the dynamic reduction factor defines the rate of the signature transfer in short loops.

Original program	`result=0;`
	`DIM=9;`
	`for (k=0; k<DIM; k++) {`
	`    result += v1[k]*v2[k];`
	`}`

Signatured (no reduction)	`*WP_BASE=0x80100401; result=0;`
	`*WP_BASE=0x00200802; DIM=9;`
	`*WP_BASE=0x00302003; for (k=0; k<DIM; k++) {`
	`    *WP_BASE=0x00401404; result += v1[k]*v2[k];`
	`} *(unsigned long *)(0x00000004)=0xC0602406;`

Static reduction (factor=5)	`*WP_BASE=0x80101001; result=0;`
	`DIM=9;`
	`for (k=0; k<DIM; k++) {`
	`    *WP_BASE=0x00501805; result += v1[k]*v2[k];`
	`} *WP_BASE=0xC0201C02;`

Dynamic reduction (factor=8)	`*WP_BASE=0x80100401; result=0;`
	`*WP_BASE=0x00200802; DIM=9;`
	`_wpc = 0; *WP_BASE=0x00302003; for (k=0; k<DIM; k++) {`
	`    if (!((_wpc++)&7)) {*WP_BASE=0x00401404;}`
	`    result += v1[k]*v2[k];`
	`} *WP_BASE=0xC0602406;`

Table 4 Reduction examples

The effects of the reduction depend heavily on the application (Table 5). In case of *linpack* the dynamic reduction reduces the number of signatures to 25%, while the effect of the static reduction is not so significant (87%). For *dhrystone* static reduction is efficient (58% and 42%) while dynamic reduction can not reduce the number of signatures further (57%). The *multigrid* solver is selected for the detailed investigations.

Number of run-time signatures		**Reduction parameters**				
		No reduction	**Statement factor 5**	**Static 5**	**Statement factor 5 Dynamic 8**	**Static 5 Dynamic 8**
Benchmark	**whetstone**	119,633 100%	76,733 64%	62,924 53%	60,320 50%	46,511 39%
	dhrystone	1,220,029 100%	710,010 58%	510,008 42%	700,010 57%	480,008 39%
	linpack	11,824,205 100%	11,121,185 94%	10,294,233 87%	2,900,598 25%	2,197,450 19%
	multigrid 5	78,992 100%	57,988 73%	35,921 45%	42,564 54%	26,262 33%

Table 5 Overview of the reduction alternatives

The run-time overhead was measured using the system timer of MEMSY (10 ms resolution), in a single-processor environment. The overhead increases drastically when using a small reduction factor, even to a level of 100% indicating a cumulative effect of multiple disadvantageous factors (Fig. 3). External bus cycles, like those required for signature transfer are by a factor of 4 to 10 slower than a cache access [5]. If there are too few statements to execute between two consecutive signatures, bus saturation can occur. In this case the CPU has to wait for the end of the transfer inactivating its internal speedup mechanisms (instruction prefetch, pipelining).

For a more detailed analysis the integrated distribution of time periods between subsequent signature transfers was measured (Fig. 4). The benchmark was executed in a single-step mode and a trace of the processor instructions was registered. In the ideal case, all signatures should be transferred within the same time period, defined by the user as a compromise between fault coverage and performance loss. The first peak in the density function (Fig. 5) after only 3 instructions results in a lesser extent from the unavoidable use of intermittent signatures in complex control structures. The dominating cause are over-tested short loops, as a costly check is performed after only a few machine instructions. Dynamic reduction or loop unrolling with a

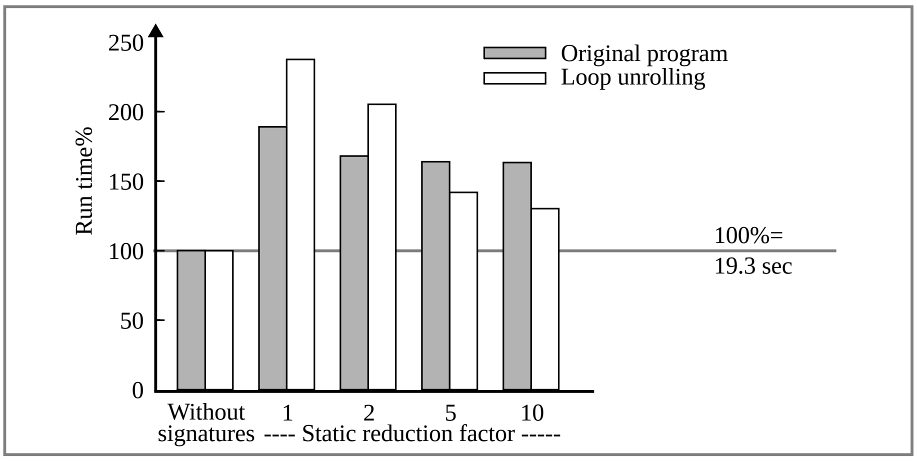


Fig. 3 Run time overhead

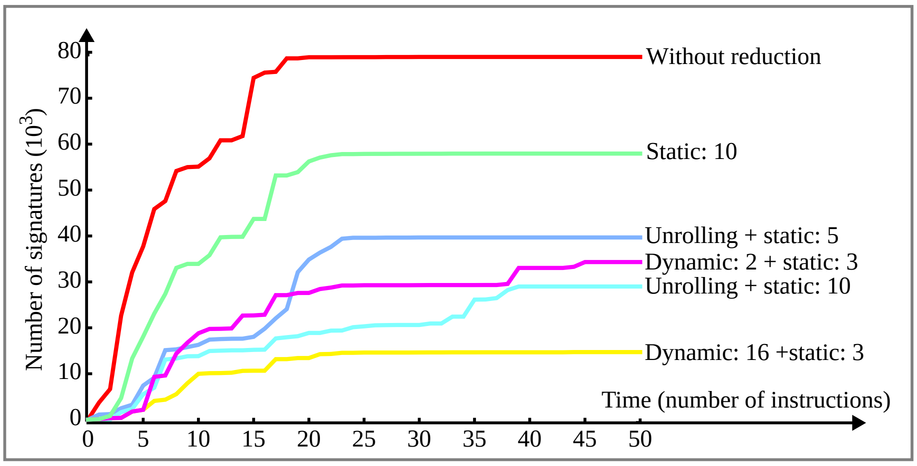


Fig. 4 Integrated number of signature transfers

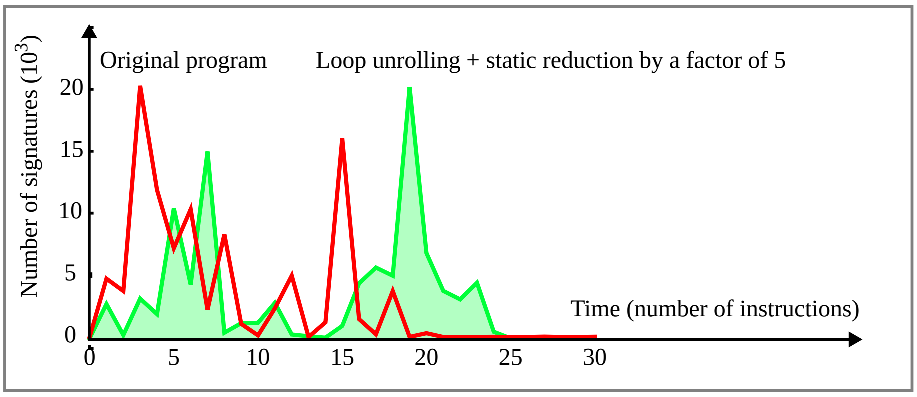


Fig. 5 Frequency density function of the time between signatures

subsequent static reduction (both puncturing signature transfers to each kth execution of the loop body) result in a radical reduction in run-time overhead without a drastic decrease in fault coverage. Undertesting can occur, even in the case of a single statement such simple as **a=b**, if **a** and **b** are complex data types involving a long copy operation.

4.2 Overhead in code length

The overhead in code length varied between 20% and 85% in the general case depending on the benchmark and static reduction factor. Results of the multigrid application are shown in Fig. 6. The number of signatures depends on the program size only approximately linearly with a moderate coefficient. This overhead is affordable even for large programs in the Mbyte range. The efficiency of the static reduction rapidly drops with increasing reduction factor. Our benchmark program consists typically of short branch-free statement sequences embedded into nested loops. (In each loop at least one signature must be included). Dynamic reduction does not influence the code length significantly.

4.3 Fault coverage

The fault coverage was measured using a software fault injector (based on the Unix *ptrace* system call, [20]). Single bit transient errors were injected into the program counter at a single random phase of the program execution. 5000 experiments were performed for each individual case; the fault coverage was estimated with a relative error less than ±5% at a confidence level

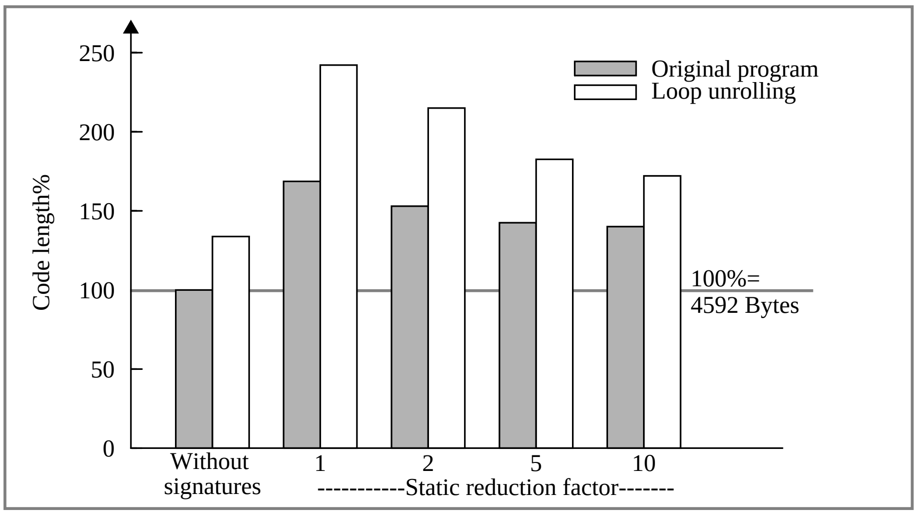


Fig. 6 Static code length overhead

of 99%. The errors masked by the program itself, affecting neither the control flow nor the final results are eliminated.

The majority of the errors was detected by the standard primary checking mechanisms of the CPU-MMU complex (segmentation violation, bus error, illegal opcode). However, for benchmarks in the Mbyte range the number of remaining errors not detected by these mechanisms increases (the probability, that a single-bit error in the address word changes one of the bits checked by the WP is depicted in Fig. 7). In the case of the extremely small *multigrid* benchmark, most of the detected errors was covered by the segmentation and bus checks (85%).

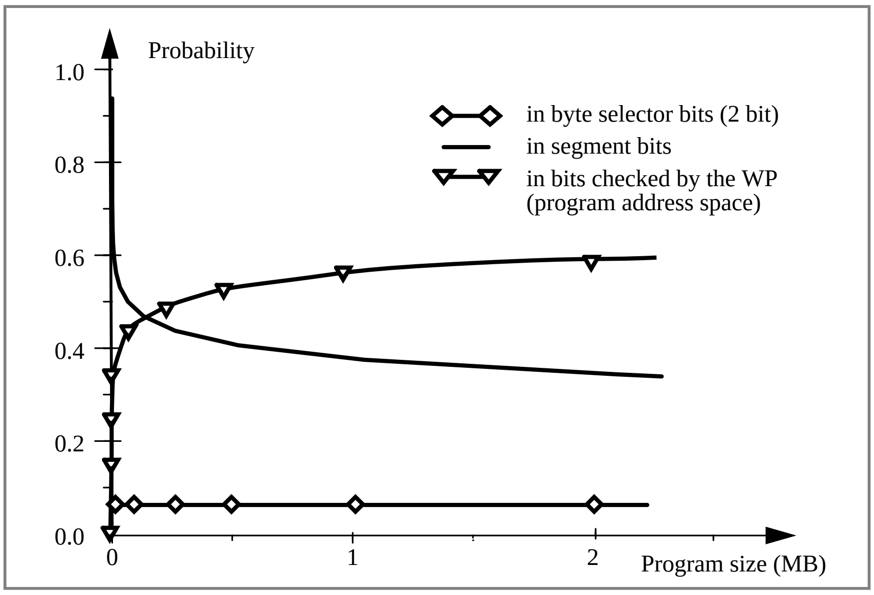


Fig. 7 Error probability in given address bits

The fault coverage (Fig. 8) of the WP is typically in the interval of 10-65% of the errors remaining undetected by the standard checking mechanisms. The decrease of fault coverage with a growing static reduction factor is a consequence of the larger address range between two consecutive signatures, as control flow errors remaining within this interval are not covered by any WP method. This overall result corresponds to the coverage of other WP implementations, like in [16].

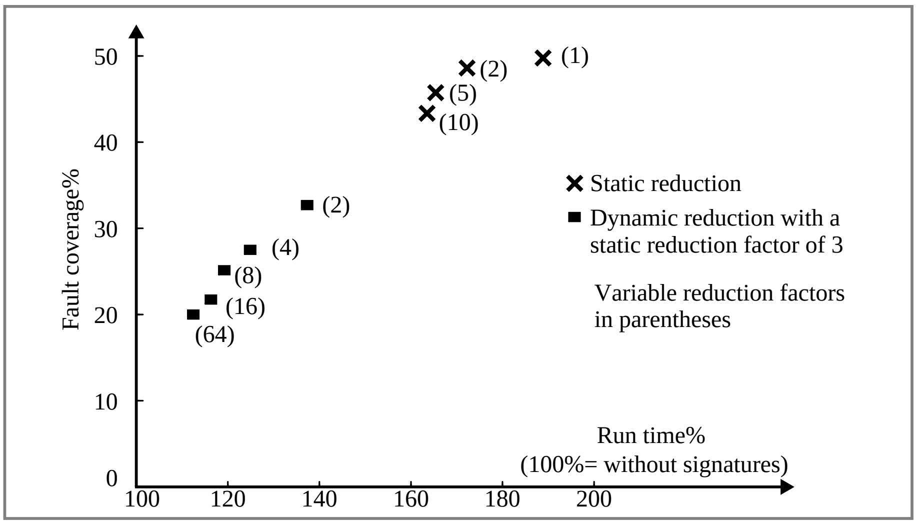


Fig. 8 Run time vs fault coverage

Conclusion

The advantages and possible use of an assigned signature watchdog processor in multiprocessor and multitasking environments were discussed. Main idea of the proposed SEIS method is the redundant encoding of the program CFG. In this way, only the last signature of each program block has to be stored as reference. The evaluation of the actual signature is a simple combinatorial task. The advantages of the proposed methods are the low hardware cost, the high processing speed and the possibility of integration into existing systems. First experiments with the MEMSY multiprocessor yielded encouraging results.

However, the traditional views on WPs based on high-level preprocessing, which originate in the very first publications on this topic, must be revised in the light of the measurement results. Beyond question, this approach remains attractive due to its outstanding advantages, like portability or compatibility with compiler-made automatic optimization. Fault coverage corresponds approximately to the known methods at the assembly level. On the other hand, the rough granularity of individual statements does not allow a sufficiently fine tuning of the distribution of signature transfers in time. The current development aims at going deeper in the syntax hierarchy down to the elementary operation level, where a similarly structured, but significantly more detailed CFG can be built as at the instruction level. When weighting the edges of this CFG with the operation execution times, the dynamic distribution of signature transfers reduces to a known optimization problem. The WP can be further used without any modification thanks to the very general and flexible nature of the encoding algorithm.

Acknowledgments

The research presented here was supported by the Hungarian-German Joint Scientific Research Project #70, Konrad Zuse Program (DAAD), OTKA-3394 and F7414 (Hungarian NSF). MEMSY was developed in the framework of the DFG (Deutsche Forschungsgemeinschaft) as part of ‘Sonderforschungsbereich 182’.

References

1 **Dal Cin, M et al**.
‘Fault Tolerance in Distributed Shared Memory Multiprocessors’, In: A. Bode, M.Dal Cin (eds.), *Parallel Computer Architectures*, LNCS 732, Springer, Berlin (1993) pp 31-48

2 **Dal Cin, M Hohl, W Hönig, J and Pataricza, A**
‘MEMSY - A Modular Expandable Multiprocessor System with Fault Tolerance’, *Proc. Parallel Systems Fair of the 8th IEEE Int. Parallel Proc. Symp.*, Cancun (1994) pp 21-28

3 **Dal Cin, M Hohl, W et al.**
‘Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY’, *Proc. 1st Int. Conf. on Massively Parallel Computing Systems* (MPCS’94), Ischia, May 2-6, 1994 IEEE (1994) pp 7-15

4 **Eifert J B and Shen J P**
‘Processor Monitoring Using Asynchronous Signatured Instruction Streams’, *Proc. FTCS-14*, IEEE (1984) pp 394-399

5 **Handy J**
The Cache Memory Handbook. Academic Press, San Diego (1993)

6 **Hofmann F et al.**
‘MEMSY - A Modular Expandable Multiprocessor System, In: A. Bode, M. Dal Cin (eds): *Parallel Computer Architectures*, LNCS 732, Springer, Berlin (1993) pp 15-30

7 **Hönig, J and Sieh, V**
‘Software-Based Concurrent Control Flow Checking’, *Internal Report IMMD 3*, Univ. Erlangen (1994)

8 **Lu D J**
‘Watchdog Processors and Structural Integrity Checking’, *IEEE Trans. Comp.*, Vol 31 No 7 (July 1982) pp 681-685

9 **Madeira H, Camoes J and Silva G**
‘A Watchdog Processor for Concurrent Error Detection in Multiple Processor Systems’, *Micro-processors and Microsystems* Vol 15 No 3 (April 1991) pp 123-131

10 **Mahmood A and McCluskey E J**
‘Concurrent Error Detection Using Watchdog Processors - A Survey’, *IEEE Trans. Comp., Vol* 37 No 2 (February 1988) pp 160-174

11 **Majzik I**
‘Fault Detection in the MEMSY Multiprocessor using a SEIS Watchdog Processor’, *Internal Re-port IMMD3 10/1993*, Univ. Erlangen (1993)

12 **Majzik I**
‘SEIS: A Program Control Flow Graph Encoding Algorithm for Control Flow Checking’, *Technical Report TUB-TR-94-EE14*, Technical University of Budapest (1994)

13 **Michel T, Leveugle R and Saucier G**
‘A New Approach to Control Flow Checking Without Program Modification’, *Proc. FTCS-21*, IEEE (1991) pp 334-341

14 **Michel E and Hohl W**
‘Concurrent Error Detection Using Watchdog Processors in the Multiprocessor System MEMSY’, In: M. Dal Cin, W. Hohl (eds): *Fault Tolerant Computing Systems*, Informatik-Fachberichte 283, Springer, Berlin (1991) pp 54-64

15 **Milner R**
Communication and Concurrency. Prentice Hall, New York (1989)

16 **Miremadi G, Karlsson J, Gunneflo U and Torin J**
‘Two Software Techniques for On-Line Error Detection’, *Proc FTCS-22*, IEEE (1992) pp 328-335

17 **Pataricza A, Majzik I, Hohl W and Hönig J**
‘Watchdog Processors in Parallel Systems’, *Microprocessing and Microprogramming* 39, North Holland (1993) pp 69-74

18 **Schuette M and Shen J P**
‘Exploiting Instruction Level Resource Parallelism for Transparent Integrated Control-Flow Monitoring’, *Proc. FTCS-21*, IEEE (1991) pp 318-325

19 **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, North Holland (1987) pp 249-269

20 **Sieh V, Pataricza A, Sallay B, Hohl W, Hönig J and Benyó B**
’Fault Injection Based Validation of Fault-Tolerant Multiprocessors’. *Proc.* µ*P’94, the 8th Symposium on Microcomputer and Microprocessor Applications,* Budapest, Hungary (1994) pp 85-94

21 **Sridhar T and Thatte S M**
‘Concurrent Checking of Program Flow in VLSI Processors’, *Proc. 1982 Int. Test Conf.*, IEEE (1982) pp 191-199