EDCC-1, the First European Dependable Computing Conference
Berlin, Germany, October 4-6, 1994

Hierarchical Checking of Multiprocessors Using Watchdog Processors

I. Majzik[++], A. Pataricza[+,++], M. Dal Cin[+], W. Hohl[+], J. Hönig [+],V. Sieh [+]

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

Abstract. A new control flow checking scheme, based on assigned-signature checking by a watchdog processor, is presented. 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 or 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 the 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 is a relatively simple coprocessor that compares the actual control flow - represented by run-time *signatures* - with the previously computed reference control flow. WPs can be used to perform other checks as well [7], like assertions on the data. The coprocessor-approach offers a 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. They can be grouped according to the way run-time signatures are generated and the source of reference. Some typical methods are presented in Table 1. The methods using *derived* run-time signatures monitor and compact 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.(In [6] 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.

Additionally, different approaches to integrate watchdog processors into multipro-

		Reference		
		Stored signature database	Watchdog program	Embedded signatures
Run-time control flow	Derived signatures	Asynchronous Signatured Instruction Stream [2]	Watchdog Direct Processing [8]	Basic Path Signature Analysis [15]
	Assigned signatures	Extended Structural Integrity Checking [9]	Structural Integ-rity Checking [4]	Signature Encoded Instruction Stream [12]

Table 1 Control flow checking methods

cessor systems are known: a Roving Monitoring Processor [13] is connected to multiple processors and monitors their states sequentially without checking their interactions. The Checker described in [5] 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 method intended to be used in multiprocessors is Extended Structural Integrity Checking (ESIC [9]). 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.

The main drawback of these methods 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 [12]). The design goals of the SEIS project were:

- An efficient *hierarchical checking* method of *multiple processors* by 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[1] (Modular Expandable Multiprocessor System [1,3]) was used as test-bed of 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. The subsequent two sections describe additional features of the watchdog processor and the integration of the SEIS WP into the MEMSY multiprocessor, respectively. The last section presents measurement results and conclusions.

1. MEMSY was developed in the framework of the DFG project SFB 182. 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)

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 (e.g. C, Pascal). Programs contain procedures composed of statements. At each level (process, procedure and statement) a different checking method and WP module is used.

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 a more formal description see [11]):

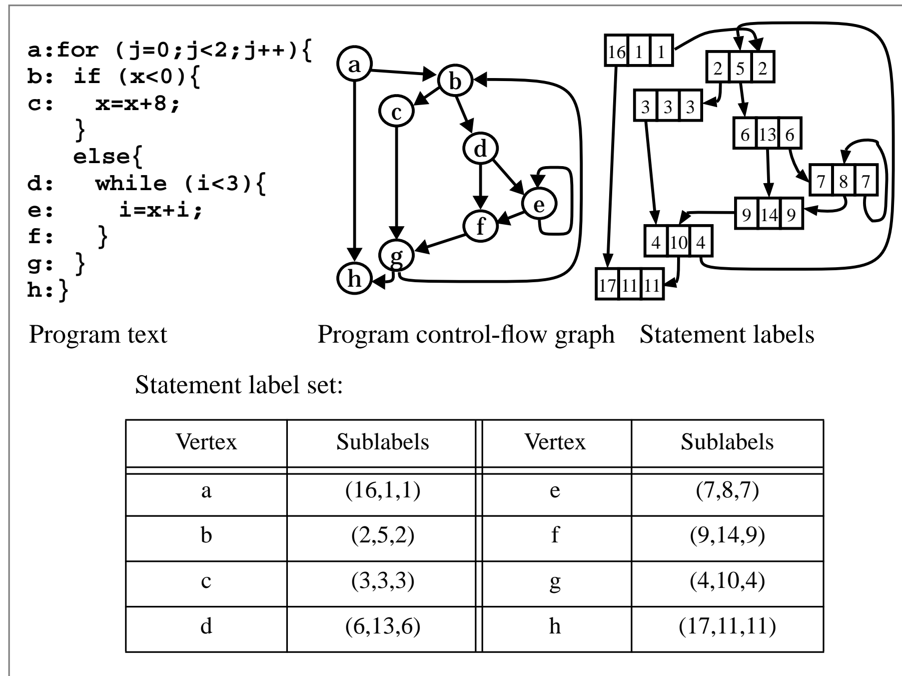
1. The *CFG* of the procedure is *extracted*. The basic control structures form subgraphs of the CFG. These subgraphs are identified according to the requirements of the encoding algorithm: that is, the number of successors of a vertex is limited in order to reduce the information to be encoded in the 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* increasing 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 [11].
   Let assume that a `case` statement with an actual sublabel of *6* has more output branches than 3, the maximal number of successor vertices allowed by the basic encoding scheme. The sublabel *7* is assigned to each successor vertex, indicating that they all are valid successors of the **case** vertex. (Note, that no data dependencies, like branch selection, are checked by the WP). The individual output branches of the case statement will be distinguished by assigning different second and third



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 The encoding of the program control-flow graph

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 result in fewer checks, increase error latency, and reduce the probability of error detection, yet on the other hand result in a shorter execution time and program size overhead. Static reduction may remove small branches in the CFG.

Removal of cycles in the CFG is not allowed, because otherwise the program may run within loops for extended periods of time without any checks. Hence, each loop has to contain at least one statement label. Overhead measurements (described in Section 5) 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 combinatorial logic circuit, the evaluation of the statement sublabels is performed by a comparator set.

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,

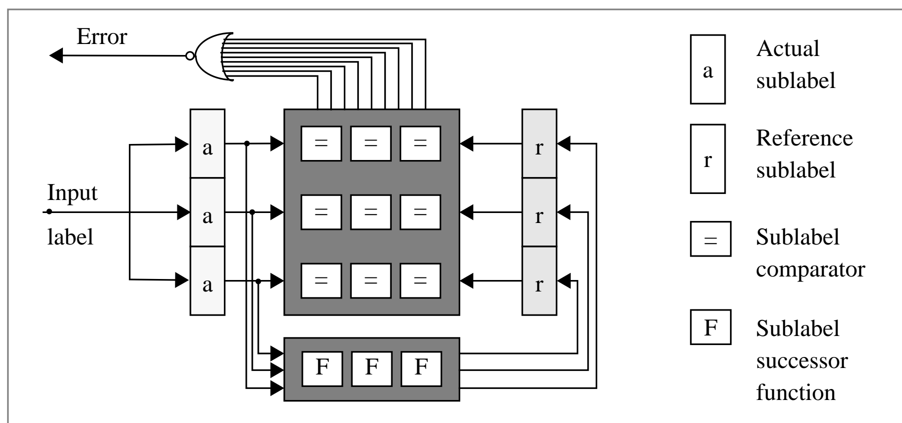


Fig. 2 Statement level checker module

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. This way 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. Nonetheless, procedure calls are allowed to start only at an entry point of a procedure, so only an erroneous jump to the starting point of a procedure can not be detected immediately.

The first and last statement labels of the 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. *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 a 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* its 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 system.

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. One major goal of the research was the extension of control flow checking to the level of interprocess cooperation. Such checks allow the blocking of the dissemination of error effects from a faulty process to the other ones. In this way the error latency and losses in computation time can be drastically reduced. For this reason a skeleton-like description of the communication and synchronization structure is required. Synchronization of application processes can be described e.g. by using a simple CCS-like process algebra [14], (Appendix).

The basic idea will be illustrated by the simplest case, the synchronous communication between two processes. This kind of communication is valid, if the sender process closes it after the receiver process has accepted the data, and the receiver can not get data before the sender begins the communication. An error is detected if one of the partner processes has already finished the communication and the other partner has not even started it. Similarly to the lower levels checks, the WP will be notified on the status of the interprocess control flow by special signatures. Reference labels are assigned to the different phases of the individual processes in order to make their state in the control flow during the interaction observable for the WP. Such a reference label should change only after a synchronization in order to distinguish the different phases during the cooperation. (It can be shown that only the synchronization statements must be guarded by special signatures).

The informal description of the checking mechanism is as follows: during the synchronisation of two processes two special signatures are sent by each process.

1. The first, *initializing signature* before the execution of the synchronization notifies the WP, with which partner process the synchronization is intended. 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.

2. The second signature is sent during the synchronization itself by each participating process. It contains the *reference signature* for the partners, computed at compilation time in the same way as the WP handles the initialization signatures. This second signature is valid only if the partner processes coincide with the expected ones, and both processes have already initialized the synchronization, otherwise an error is detected.

A register (called *reference register*) is reserved for each process internally in the WP. The synchronization checker module executes two operations depending on the actually received guard signature:

1. *Initialization*. The initializing signatures before the statement of synchronization contain the ID of the presently running process and the ID of the intended partner for communication. 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, than the process is the one beginning the synchronization and the partner is uninitialized. To indicate this, the checker module stores the ID of the partner in the reference register of the running process (unidirectional, actual $\rightarrow$ partner process initialization).
   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 the communication is allowed and the partners are ready for it.
2. *Checking*: The second guard signature (after the synchronization statement) transfers the synchronization label, which was computed by the precompiler using the same function as the one used internally in the WP for the generation of the reference label. The WP checker module compares this label with the content of the reference register computed during the initialization step. Upon a mismatch an error exception is raised.

In order to support the checking mechanism above, the precompiler has to parse the program text to identify the processes to be synchronized at a given statement, generate the initializing signatures and the synchronization labels.

The examples in Table 2 present the insertion of guard signatures for synchronous communication (the sender has to wait for the receiver) and fork-join structures. The

Example		Checked process system[a]	Remark
Communication (synchronization)		$I(A,B).\overline{data}.C(A\otimes B).Ps$ / $I(B,A).data.C(B\otimes A).Pr$	A, B: process IDs Ps: sender (cont'd) Pr: receiver (cont'd)
Sender	Receiver		
$\overline{data}.Ps$	$data.Pr$		
Single fork (without join)		$I(A,B).\overline{fork}.Pp$ / $fork.I(B,A).C(A\otimes B).Pc$	A,B: defined values Pp: parent (cont'd) Pc: child activities
Parent	Child		
$\overline{fork}.Pp$	$fork.Pc$		
Fork-join structure		$I(A,B).\overline{fork}.exit.I(A,B).C(A\otimes B).Pp$ / $fork.I(B,A).C(B\otimes A).Pc.I(B,A).\overline{exit}.0$	A,B: defined values Pp: parent (cont'd) Pc: child activities
Parent	Child		
$\overline{fork}.exit.Pp$	$fork.Pc.\overline{exit}.0$		

[a] I(A,B) means that the process initializes the checking with A as actual and B as partner label.
C(A⊗B) means the initialization of the checking of label A⊗B using the reference label.

Table 2 Checking of the synchronization

UNIX-style *fork()* system call creates a child process by duplicating the parent. The parent may wait for the termination of the child (using the *wait()* system call, which means a fork-join structure) or the two processes can run independently. These examples are asymmetrically checked, since an initialization guard can not precede the starting point of the child process, and, the child can not check its own termination (Table 2). In the single fork structure, it is assumed that the parent does not make a new synchronization before the child starts. Process algebra provides a compact description of the process interaction structure, as illustrated by the examples in Fig. 3.

Process level checking uses as a hardware resource only the *processor-process database* (one record per processor) and the *reference registers* (their number is proportional with the number of processes).

2.4 Checking Hierarchy

In summary, the following types of information are combined to form 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 actual signature;
- *synchronization labels* (special guard signatures).

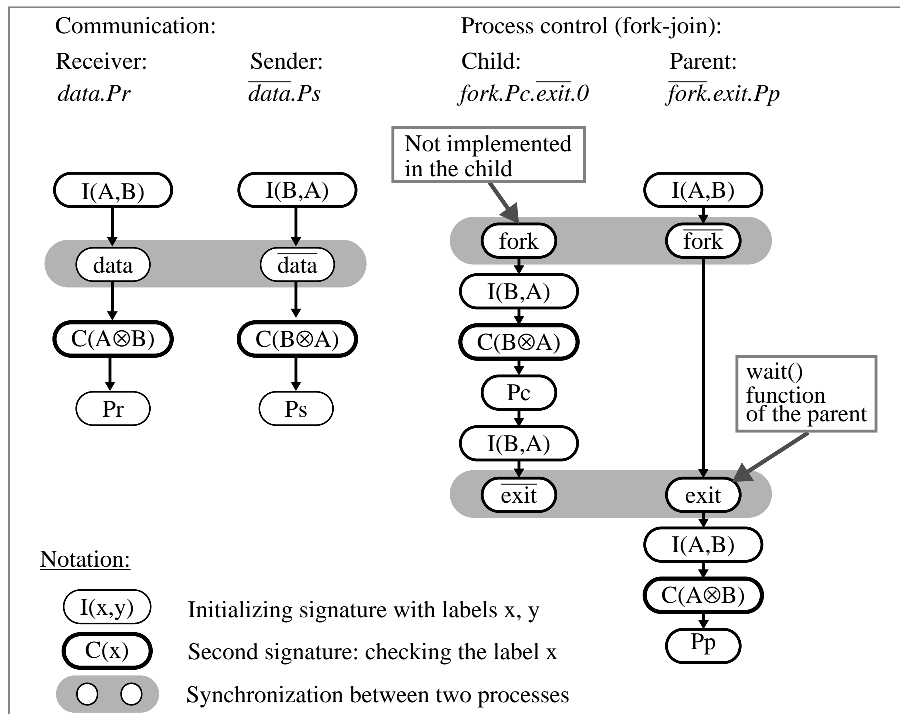


Fig. 3 Examples of the process synchronization

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 3. 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. The check of process scheduling provides an additional method to detect faults in the process descriptors and pointers to the process table.

Checker level	Checked operation	Signature information	Checker method
Statement	Statement sequence	Statement labels	Comparison
Procedure	Call and return	Procedure ID	Ref. 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 3 Hierarchical checking (summary)

3 Additional Features of the Watchdog Processor

3.1 Error Recovery

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. Two-phase commit is used in order to always have a valid checkpoint. Each process stores its state as a tentative checkpoint in one of two buffers. A tentative checkpoint is made permanent, if all processes succeed in taking their checkpoint. Otherwise, the system restarts from the previous permanent checkpoint.

Checkpointing and restarting a process requires the WP to save and restore its 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. Thus, checkpointing only requires saving the operational stack pointer, and write-protecting the ref-

erence 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.

3.2 Error Notification

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. An internal status FIFO is used in the WP in order to avoid error signal overruns.

4 Integration of an Experimental SEIS WP into a Multiprocessor

4.1 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. 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 check simultaneously 5 computing nodes consisting of a total of 20 processors and running a maximum of 1280 processes [10].

4.2 Signature Transfer

Different types of information have to be transferred to the WP, as pointed out in the description of the checker modules. WP accesses can be grouped as *commands* (e.g. checkpoint generation, recovery, initialization), *special signatures* (synchronization guards, scheduler informations) and *common signatures*. In order to simplify the communication with the checked system, the WP was implemented as a *multiport coprocessor* with ports mapped into the address ranges of the checked processors. Commands and signatures are transferred through shared memory write cycles; status information can be obtained by a read cycle. In order to achieve a sufficient bandwidth during signature transfers, the address word of the signature write access is used to transfer signature information as well (Table 4).

	Address (word addressing mode)				Data			
	Sig. type	Processor ID	Process ID	Procedure ID	Flags	Sublabel #1	Sublabel #2	Sublabel #3
Bits	2	2	8	10	2	10	10	10

Table 4 Signature structure

As example, the preprocessed internal loop of the program in Fig. 1 is shown below. Static reduction is disabled. The procedure ID in this example is *1*, process and processor IDs are *both 0*. The assumed base address of the WP signature register is hexadecimal *B4000000*. The program uses byte addressing mode, thus a byte address offset of the value of 4 corresponds to a word offset of 1.

```
*(unsigned long *)(0xB4000004)=0x00603406;
  {while (i<3) {
    *(unsigned long *)(0xB4000004)=0x00702007;
    {i=x+i;}
  } *(unsigned long *)(0xB4000004)=0x00903809;}
```

The internal loop is modified by defining *16* as dynamic reduction factor to:

```
_wpc=0; *(unsigned long *)(0xB4000004)=0x00603406;
  {while (i<3) {
    if (!((_wpc++)&15)) {
      *(unsigned long *)(0xB4000004)=0x00702007;}
    {i=x+i;}
  } *(unsigned long *)(0xB4000004)=0x00903809;
```

4.3 Shared Operation

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

using a round-robin priority scheme. Signature checking is executed 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.

4.4 MMU Utilization

As described earlier, the process ID field is embedded into the part of the signature transferred via the address bus. During preprocessing and compiling, the same constant values are assigned to all process ID fields in the statement labels, because these IDs are unavailable at compilation time. At run-time, depending on the process ID, the same virtual address ranges are mapped by the MMU to different physical address ranges of the WP. Thus, the unrestricted use of *shared code* and *shared libraries* is supported.

In the MMU WP address sub-ranges can be defined as nonexistent or write-protected. Only a single page corresponding to the process ID is visible in the user address space of a process preventing illegal accesses to the address range of other processes. Additionally, WP commands (e.g. checkpointing and recovery) are privileged, avoiding, for example, an accidental checkpoint overwrite.

4.5 Implementation Details

SEIS Precompiler. The current precompiler processes programs written in C language. The precompiler itself was written in C too, so it is fully portable to different platforms. For parsing *bison* is used, the encoding algorithm is a linear one.
WP Hardware. The WP was implemented as a 16MHz 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 256K RAM block which proved to be oversized if no recursive programs were 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.
Operating System Modifications. The operating system of the checked computers was only slightly modified. Interrupt handlers serving the WP (e.g. detected error, saturation of the WP stack) have to be added to the system. The creation or scheduling of a new process requires the initialization of the internal WP processor-process database and the initialization of the address translation tables in the MMU.

5 Measurement results

Standard benchmarks (like *dhrystone, whetstone, 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. Compilation was done using *gcc version 2.2.2* generating a highly optimized code. The following characteristics were measured for each reduction strategy:

- *static code length*;
- *fault coverage* by injecting single bit transient errors 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 of 99%);
- *program run times*, using the system timer of MEMSY with a resolution of 10 msec;
- *number of signatures* sent to the WP;
- *distribution of time between subsequent signature transfers* in terms of number of instructions executed by tracing the program in single-step mode.

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 by the detailed results of the multigrid solver. The experiences with other benchmarks will be presented as accumulated intervals.

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. 4. 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 a growing reduction factor. Our benchmark program consists typically of short

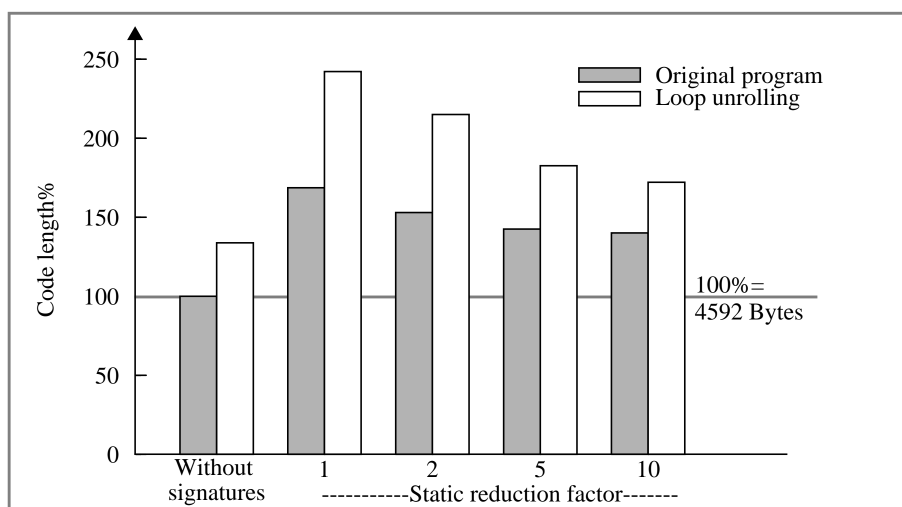


Fig. 4 Static code length overhead

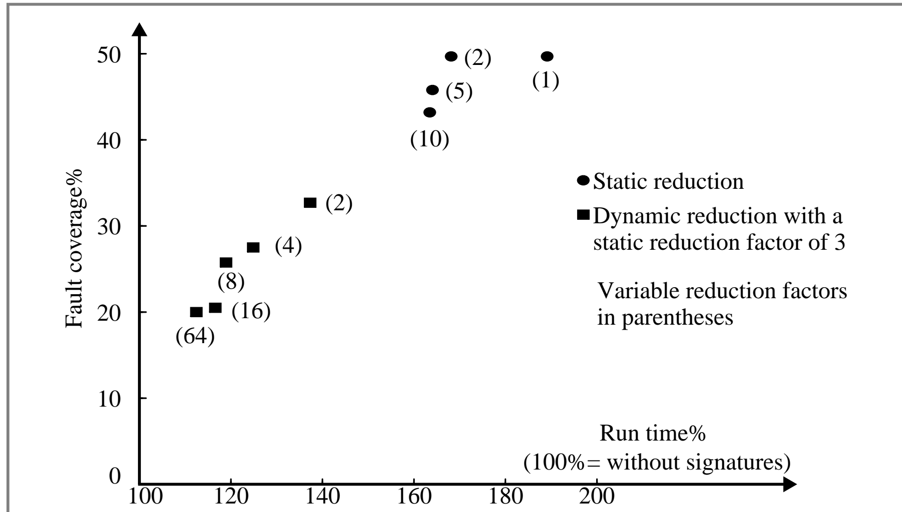


Fig. 5 Run time vs fault coverage

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.

Fault coverage (Fig. 5) is typically in the interval of 10-65% of the errors remaining undetected by the standard primary checking mechanisms of the CPU- MMU complex (nonexisting address, illegal opcode etc.). In this figures the errors masked by the program itself, affecting neither the control flow nor the final results are eliminated. 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].

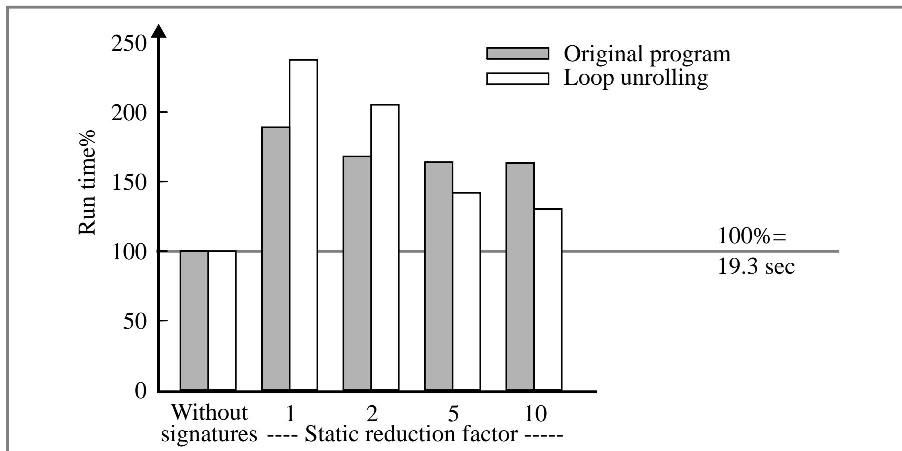


Fig. 6 Run time overhead

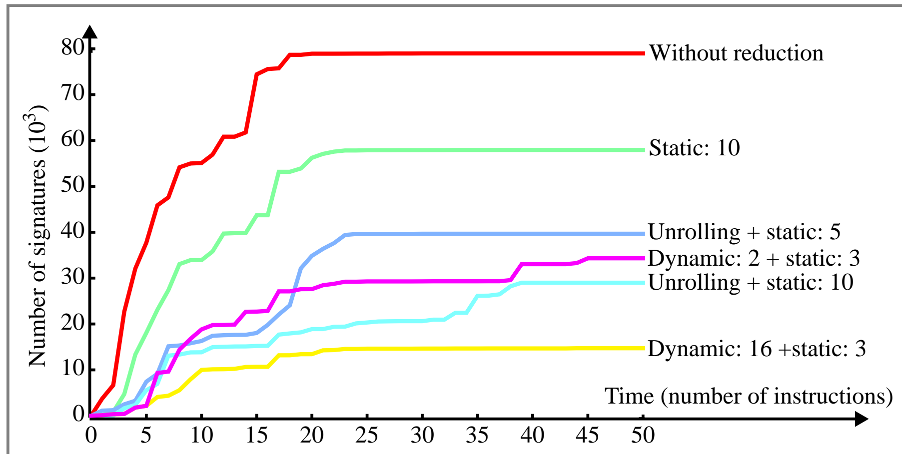


Fig. 7 Integrated number of signature transfers

The run-time overhead increases drastically when using a small reduction factor, even to a level of 100% indicating a cumulative effect of multiple disadvantageous factors (Fig. 6). External bus cycles, like those required for signature transfer are by a factor of 4 to 10 slower than a cache access [17]. 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. 7).

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. 8) after only 3 instructions results in a lesser extent from the unavoidable use of intermittent signatures in complex control structures. The dominating cause are overtested short loops, as a costly check is performed after only a few machine instructions. Dynamic reduction or loop unrolling with a subse-

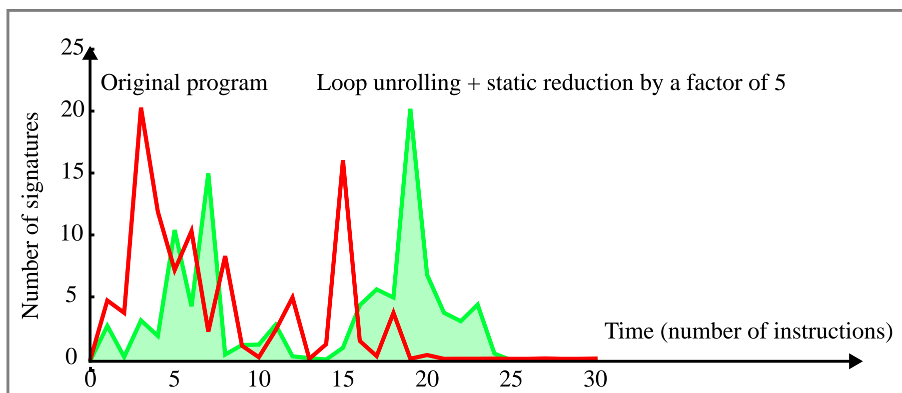


Fig. 8 Frequency density function of the time between signatures

quent 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.

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 easy 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.

Appendix

The elements of the process algebra are defined as follows:

- $P,Q,...$ *agents* (representing the processes);
- $a,b,... \in L$ *labels* (representing observable operations e.g. receive a and send $\bar{a}$, respectively; 0 represents the end of the process);
- $\alpha,\beta,... \in Act$, $(Act = L \cup \{\tau\})$ *actions* (e.g. send, receive and the internal synchronization τ).

The expressions are composed with the help of three operators as prefix (., sequencing of actions), summation (+, non-deterministic choice) and composition (/, parallel execution): $P ::= 0 \mid \alpha.P \mid P+P \mid P/P$.

References

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

2 J.B. Eifert, J.P. Shen: Processor Monitoring Using Asynchronous Signatured Instruction Streams. Proc. FTCS-14, 394-399 (1984)

3 F. Hofmann et al.: MEMSY - A Modular Expandable Multiprocessor System. In: A. Bode, M. Dal Cin (eds): Parallel Computer Architectures. Lecture Notes in Computer Science 732, Berlin: Springer, 1993, pp. 15-30

4 D.J. Lu: Watchdog Processors and Structural Integrity Checking. IEEE Trans. on Comp. 31, 681-685 (1982)

5 H. Madeira et al.: A Watchdog Processor for Concurrent Error Detection in Multiple Processor Systems. Microprocessors and Microsystems 15, 123-131 (1991).

6 M. Schutte, J.P. Shen: Exploiting Instruction Level Resource Parallelism for Transparent Integrated Control-Flow Monitoring. Proc. FTCS-21, 318-325 (1991)

7 A. Mahmood, E.J. McCluskey.: Concurrent Error Detection Using Watchdog Processors - A Survey. IEEE Transactions on Computers 37, 160-174 (1988)

8 T. Michel, R. Leveugle, G. Saucier: A New Approach to Control Flow Checking Without Program Modification. Proc. FTCS-21, 334-341 (1991)

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

10 I. Majzik: Fault detection in the MEMSY multiprocessor using a SEIS watchdog-processor. Internal report 10/1993 of the IMMD3, Universität Erlangen, 1993

11 I. Majzik: SEIS: A program control-flow graph encoding algorithm for control flow checking. Internal report, Technical University of Budapest, 1994 (in Press)

12 A. Pataricza, I. Majzik, W. Hohl, J. Hönig: Watchdog Processors in Parallel Systems. Microprocessing and Microprogramming 39, 69-74 (1993)

13 J.P. Shen, S.P. Tomas: A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems. Microprocessing and Microprogramming 20, 249-269 (1987)

14 R. Milner: Communication and Concurrency. New York: Prentice Hall, 1989

15 T. Sridhar, S.M. Thatte: Concurrent Checking of Program Flow in VLSI Processors. Proc. 1982 Int. Test Conf., 191-199 (1982)

16 G. Miremadi et al.: Two Software Techniques for On-Line Error Detection, Proc FTCS-22, 328-335 (1992)

17 J. Handy: The Cache Memory Handbook. San Diego: Academic Press 1993.