A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog

Joakim Ohlsson, Marcus Rimén, Ulf Gunneflo

Department of Computer Engineering, Chalmers, Sweden

ABSTRACT

This paper presents an error-detecting 32-bit RISC, designed in a 1.2μm CMOS technology, with an on-chip watchdog using embedded signature monitoring. It is evaluated through simulation-based fault injection, using a register level model written in VHDL. A chip area increase of 4.7% was caused by the watchdog. Two application programs were executed to study workload dependencies. The insertion of watchdog instructions resulted in a memory overhead of between 13% and 25% as well as a performance overhead of between 9% and 19%. A total of 2,779 faults were injected into the processor during execution of the application programs. Only 23% of these resulted in effective errors. A minimum detection coverage of 95% was reached for effective errors classified as control flow errors with a median latency of 1 clock cycle. Few effective data errors, between 22% and 50%, were detected.

Key words: *concurrent error detection, error detection coverage, error detection latency, transient fault injection, watchdog processor, control flow monitoring, signature analysis, embedded signature monitoring.*

1 Introduction

The need for employing concurrent error detection in processors will probably increase in the future as device geometries decrease. Devices will become more susceptible to transient faults because of increasing clock frequencies and lower energy differences between logic levels [16]. At the same time, more processors are subjected to noisy environments, as computers are becoming more common in automobiles, public transportation systems, process industry, etc.

Replication of the entire processor with a check on the pair-wise result is, while effective, not sufficiently economical for many applications. The redundancy can be less than 100% if techniques such as self-checking circuits or various coding techniques, e.g. parity or residue codes, are applied to processor substructures [4], [13].

Other techniques are based on monitoring abstractions of the system behavior. The devices used for this monitoring are known as watchdogs or monitors. Behavior-based error detection techniques have a potential for high error detection coverage using a simple watchdog. The watchdog collects information about the system concerning, for example, the memory access behavior, the control flow or the control signal assertions, and checks that the behavior is consistent with the selected abstraction. Many computing systems are provided with simple watchdog mechanisms such as illegal opcode detection or detection of illegal bus accesses. These types of simple monitors are however not sufficient in many applications. Experiments reported by Gunneflo *et al.* [6] indicate that control flow errors account for the majority of the errors caused by transient faults. Monitoring of abstractions of the control flow therefore have a high error detection potential. Many of the recently proposed techniques for control flow monitoring use a comparatively simple watchdog and signatured programs [3], [9], [11], [12], [14], [15], [17], [18], [22].

In signature monitoring, the watchdog computes run time signatures by compacting invariant abstractions of the processor behavior, e.g. sequences of instruction words. At some specific points, the watchdog compares the run time signatures with precomputed reference signatures. Sequences of signals from any control level that produce a constant signature can be monitored: machine code, control lines or microcode. In this paper, however, only the machine code level is considered. The reference signatures are generated by the compiler and can either be stored in a special memory reserved for the watchdog [12], [19], or be embedded into the machine code [11], [14], [18].

The use of on-chip caches, pipelining and delayed branches increase the complexity of off-chip signature monitors for modern microprocessors, as indicated by Delord *et al.* [2]. A solution that reduces the hardware overhead considerably is the integration of the watchdog

with the microprocessor itself. On-chip signature monitors have been proposed by several authors [8], [18], and Leveugle *et al.* [8] have shown that an on-chip signature monitor can be implemented with a small increase in complexity for a non-pipelined processor. However, no on-chip implementation of a signature monitor has been reported for a processor utilizing techniques such as pipelining and delayed branching, which are used in most modern processor designs.

This paper presents a conventional five-stage pipelined 32-bit RISC processor, called TRIP-Test RIsc Processor, with built-in data and control flow error-detecting mechanisms. It is evaluated by simulation-based fault injection, using a register level model written in VHDL. The processor was originally designed for use in the nodes of a vehicular communication network. However, the results should be applicable to similar RISC designs. Studies of dependable computer networks in vehicles [20] indicate that computer nodes in such networks should have fail silent behavior for permanent faults and also be able to perform fast recovery from transient faults. As the intended applications can be expected to demand hard real-time requirements on execution performance and context switching (interrupt handling), transient faults should be detected within as few clock cycles as possible. TRIP can detect a number of errors arising from abnormal program execution such as bus errors, address errors, privilege violations and illegal opcode fetches, which all cause the generation of an exception. These exceptions are referred to as the basic error-detecting mechanisms. In addition, data errors are intended to be detected by a parity-checking technique, used extensively on the internal registers of TRIP, while a built-in watchdog is intended to detect control flow errors. The error detection coverage of the parity-checking method is not evaluated in this paper, as the coverage obtained by injecting faults into registers can be predicted to be higher than the coverage of real faults. In the simulation model, the coverage is 100% for faults injected into registers checked by parity while, in a real system, faults can affect combinatorial networks as well as registers.

2 Error Detection Methods

2.1 Watchdog Method

The watchdog (WD) uses signature analysis to detect deviations from the intended program flow. The signature analysis method is based on basic signature-monitoring techniques that store signatures within the processor's program space [11], [14], [18]. Such techniques are termed Embedded Signature-Monitoring (ESM) techniques [21]. ESM is here modified to take advantage of the pipelined architecture of a RISC. A program using this modified ESM is partitioned into **branch-free intervals** and **branch instructions**. The beginning of a branch-free interval may be the destination of a branch instruction or the start of, for example, an interrupt handler, both referred to as **branch-in points**, or the instruction following a **branch-out point**. A branch-free interval is ended by either a branch instruction, called a branch-out point, or by a branch-in point. A **basic block** is only a branch-free interval if the interval is ended by a branch-in point. Otherwise, it is the branch-free interval and its following branch instruction. Each basic block has an associated embedded signature which is based on the bit-patterns of the basic block's instructions. The basic block and the embedded signature constitute a **checked block**. The **basic block size** is defined as the number of instructions n the block. The **checked block size** equals the basic block size + 1. The signature is determined at link time and embedded in the code. During program execution, a run time signature is computed and compared to the embedded reference signature. If a discrepancy is found, either because the executed instructions were not the intended ones or they were not executed in the correct sequence, the watchdog signals an error. The **intermediate signature** of an interval is the signature based on the instruction words of the interval. The signature is computed using a cyclic redundancy code with the generating polynomial $h(x) = x^{16} + x^9 + x^7 + x^4 + 1$. A zero value is used as seed for the signature monitoring of a basic block.

The hardware used for signature monitoring is restricted to a Parallel-input Linear Feedback Shift Register (PLFSR) to hold the signature and an instruction counter. On exiting a basic block, the PLFSR is cleared and the instruction counter is initialized with the size of the largest immediately following basic block. If, for

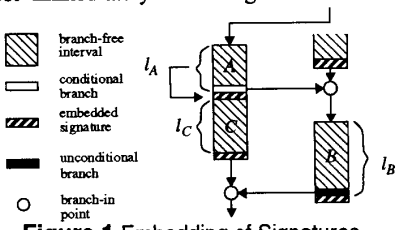

Figure 1 Embedding of Signatures.

example, the basic block A in Figure 1 has just been executed, the instruction execution may, after the intermediate signature has been compared to the embedded signature, continue by taking a branch to the basic block B or by executing the basic block C. If the size of basic block B, l_B, is greater than the size of basic block C, l_C, the instruction counter will be initialized to l_B. For every instruction executed, the counter is decremented and the signature register is updated with a value based on the bit-pattern of the instruction. When the end of a block is reached, the value of the PLFSR is compared with the embedded reference

signature and any discrepancy is signalled. Should a reference signature not be reached before the instruction counter reaches zero, an instruction counter timeout is signalled. The comparison is performed using a dedicated instruction, a `tst` instruction, which holds the reference signature, as well as the new value for the instruction counter. Because the processor uses pipelining, the instruction following a branch is always executed independent of whether the branch is taken. The `tst` instruction can therefore be inserted in the delay slot following the branch. This will give an embedding of reference signatures as shown in Figure 1. A `tst` instruction is always inserted after a branch and before a branch-in point. This is also the case for subroutine call instructions, but the instruction counter must be saved on entering the subroutine and restored on exiting. If a delay slot is normally occupied by a `nop` (no operation) instruction, no performance overhead will be introduced when a signature is embedded.

If an instruction flow error leads to a branch to unmapped/unused memory not containing the opcode of the `tst` instruction, this will be detected by the instruction counter timeout. A branch to unmapped/unused memory preceding a basic block should be detected by the signature comparison following the block, as the accumulated signature most likely will be incorrect. Assuming that bit-patterns '0...0' and '1...1' are common in unmapped/unused memory, and that the size of a word is reduced to half by bit-wise XORing the bits of the word before applying it to the PLFSR, only zeros will be accumulated to the signature for memory words with these patterns. If the signature is initialized to zero, e.g. after a branch is taken, the zero bit-pattern applied to the PLFSR will leave the signature at zero. Thus, if an illegal branch is made to unmapped/unused memory preceding a basic block, this will not be detected as the new block uses a zero signature seed. To avoid this, one-half of the bits of the instruction word were inverted before bit-wise XORing and application of the resulting bit-pattern to the PLFSR.

This section presents reasons for not using branch-address hashing (BAH) [15] and justifying signatures (JS) [11], [21]. BAH was not considered a good alternative as it is not always possible to determine the maximum number of instructions executed before an embedded signature is encountered. Furthermore, the pipelined architecture of TRIP makes it difficult to achieve good performance for BAH. A technique using JS, as described by Wilken *et al.* [21] may be used to overcome problems with correlation between initial intermediate signatures by randomizing them for the basic blocks of a program. As for BAH, the pipelined architecture of the processor makes it difficult to add a JS to a branch, as the instruction after the branch always is executed. Furthermore, with JS, there is also a problem of guaranteeing a maximum latency for error detection. Nevertheless, JS can be used for unconditional branches to decrease the number of correlated intermediate signatures. The use of JS in such cases is limited by the fact that the destination address of the branch may not always be possible to determine at compile time if the address computation is related to a register value. JS was not used for the above reasons.

2.2 Basic Error-Detecting Mechanisms

The four basic error-detecting mechanisms (BAS) all generate an exception when an error is detected. The conditions for which a specific exception is generated are described below.

An **address error** (AE) exception is generated internally as the result of a branch or data access to an unaligned address. A **bus error** (BE) exception occurs when an external device terminates a bus cycle with a bus error signal. Bus error can be asserted to indicate bus accesses to unoccupied addresses and write accesses to addresses occupied by ROM. An **illegal opcode** exception occurs whenever an instruction is encountered containing an unused opcode bit pattern. In order to provide system security, some instructions are privileged in the sense that they may not be executed while in user mode, as opposed to supervisor mode. Any attempt to do so will generate a **privilege violation** exception.

3 Implementation Issues

3.1 Description of TRIP

TRIP is a 10 MHz 32-bit RISC with five logical pipe-stages. A four-phase clock scheme (F_1 to F_4) is used to synchronize the operation of the five logical pipe-stages. As the pipe-stages are alternately latched by F_1 and F_3, data propagates through two pipe-stages for each clock cycle, which explains why only one delay slot is generated by branch instructions. All instructions use a constant width 32-bit format, which includes operands stored in the instruction field. They are executed with an executional throughput of one instruction per clock cycle for a majority of the instructions. Memory-accessing instructions are executed in two (load) and three (store) cycles respectively. TRIP has thirty-one 32-bit user registers, stored in a register file.

The instruction set can be divided into arithmetic instructions, memory-access instructions, branch instructions, surprise-related instructions and watchdog instructions. Surprise-related instructions support system traps, i.e. software interrupts, and context switching.

Figure 2 shows the structure of the data path of TRIP. The five pipe-stages can be distinguished: instruction fetch

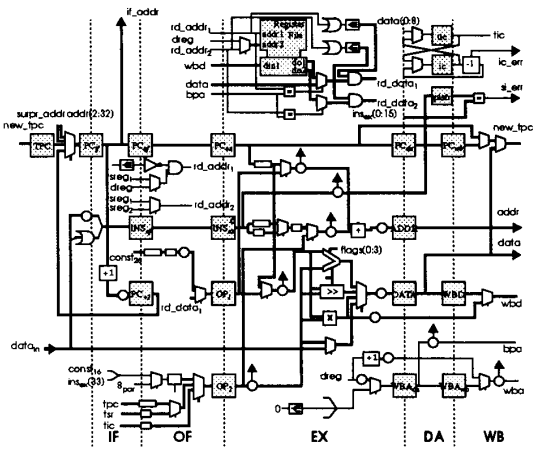

Figure 2 Data Path of TRIP.

(IF), operand fetch (OF), execute (EX), data access (DA) and write back (WB). In IF, an instruction fetch is initiated. The OF stage fetches operands from the register file to be used by the EX stage. In the EX stage, an arithmetic operation is performed on the operands. If the executed instruction is a memory-access instruction, the memory access will be performed by the DA stage. If not, the result is merely passed to the WB stage, where data is written to the register file. The watchdog contains a signature register (PLFSR) and an instruction counter. It updates the signature register based on the bit-pattern of the instruction register in the EX stage whenever a new instruction is latched into the EX stage. At the same time, the instruction counter is decremented.

Events that can alter the normal program flow are called surprises. A surprise is either a trap (software interrupt), an exception, an external interrupt or a reset of TRIP. Exceptions were described earlier in the section *Basic Error-Detecting Mechanisms*. An external interrupt can, if interrupt processing is enabled, generate a surprise. However, the surprise will not be serviced until an embedded signature is encountered. This is to ensure that there is no pending control flow error. The processor will enter the fail state should an exception occur when the processor is in the progress of performing a context switch owing to a previous surprise.

The watchdog uses embedded signature-monitoring supported by four instructions: `tst`, `swapic`, `save` and `restore`. `tst` terminates the preceding test sequence by performing a check on the resulting signature; at the same time, the next test sequence is initiated. `tst` has two operands: the number of instructions executed since the previous `tst` instruction, `<prev-tlen>`, and the maximum number of instructions to be executed until the next `tst` instruction should be encountered, `<new-tlen>`. The reference signature is a part of the `tst` instruction, but need not be given as an operand, as the reference signature is computed by the linker at compile time. A `tst` instruction is inserted into the program code immediately after each basic block of the program, as described previously. However, the usage of the `tst` instruction is changed slightly when returning from a subroutine call. When calling a subroutine, the intermediate signature is cleared by a following `tst` instruction. The `<new-tlen>` operand of the `tst` instruction should be equal to the block size of the basic block following the subroutine call instruction. The `swapic` instruction is used by the subroutine to save this block size into the temporary instruction counter, `.tic`, and to initialize the instruction counter with the block size of the first basic block of the subroutine. The saved block size is later restored by the `tst` instruction following the 'return from subroutine' instruction. As subroutines may be nested, it is necessary to provide mechanisms for saving/restoring the temporarily saved instruction counter to/from the external memory. The instructions `save` and `restore` are used for this purpose. `save`, `restore` and `swapic` are termed **watchdog-supporting instructions**, as they do not deal with the actual comparison of the reference signature.

As a special case, the instruction counter is automatically initialized to a fixed maximum value, 255, when a surprise occurs, as the test length of an arbitrary surprise handler cannot be determined.

3.2 Hardware Overhead as a Consequence of the Built-In Watchdog

The SOLO 1400 system was used to assess hardware requirements in terms of chip area and the maximum operating frequency of TRIP when implemented in a 1.2μm CMOS technology. SOLO uses the notion 'transistor stage' to refer to a CMOS transistor pair. It was found that the inclusion of watchdog-related logic increased the number of transistor stages by 4.7% and that the maximum operating frequency, 10 MHz, was unchanged. The reason for using the 'transistor stage' notion as basis for comparison was that the actual chip size increased(!) when the watchdog and its related logic was removed from TRIP, as a consequence of the placement and routing strategy of the SOLO system.

The hardware overhead of 4.7% is slightly less than the 6.5% overhead obtained as the result of another study of the implementation of on-chip monitoring hardware in a 32-bit processor, HSURF_32 [8].

4 Experimental Evaluation

4.1 Experimental setup

A VHDL model of TRIP was manually extracted from

the SOLO design. A computer system was designed in VHDL consisting of TRIP, 1 MByte ROM, 16 MByte RAM and necessary glue logic. An application program was stored in the ROM and the system stack and variables were placed in RAM. It was essential for the error detection coverage evaluation of the bus error and watchdog mechanisms that the size of the memories in the system were realistic.

The objective of the experiment was to simulate TRIP as part of a computing system, called the test system, in order to study its behavior in the presence of transient faults. Simulations were started from an internal system state corresponding to a reset of the test system. Two types of simulations were performed: reference and fault injection simulations. A reference simulator simulated the fault-free operation of TRIP executing an application program, while a fault injection simulator simulated the injection of a single fault into a randomly chosen state element bit in TRIP as well as the behavior of the test system for 100 system clock cycles following the fault injection. The time between two fault injections, expressed in system clock cycles, was randomly chosen with a uniform probability distribution in the range from 1 to 19. Recordings of reference and fault injection simulations were later compared in the analysis phase of each experiment.

Faults were produced by toggling the value of a randomly chosen **internal state element bit** of TRIP, i.e. any bit of the register file or any internal d-latch or sr-latch. Only single-bit transient faults were injected. The fault duration was 1 clock cycle in order to ensure that the fault would manifest itself as an error for at least one cycle.

The execution of two application programs, called PILOT and HSORT and written in the C programming language, was simulated. **PILOT** was originally written in Ada and used as a benchmark program by Saab Space AB to evaluate the performance of RISCs for space applications [1]. It was converted from Ada to C and modified to use integer instead of floating-point arithmetic. It performs a set of operations, which is representative for an Ariane launcher piloting program. **HSORT** is a C implementation of the heap sort algorithm, identical to the one used in physical fault injection experiments with the M6809E [6]. The program sorted an array of pointers to 50 data records according to the value of a 16-bit integer variable located at the beginning of each record.

As the instruction set of TRIP is different from other processors, a complete set of tools for developing C and assembly programs was developed. Assembly language programming can be done using either MIPS Core [5] or TRIP instructions. The programs below constitute the available software development tools. **mcc** (Mips C Compiler) generates MIPS Core assembly code. **mtt** (Mips To Trip) is a filter which translates MIPS Core assembly language programs to the corresponding TRIP code. **awi** (Add Watchdog Instructions) is a program which analyzes TRIP assembly language programs and inserts watchdog instructions. **tas** (Trip ASsembler) translates TRIP assembly language programs to relocatable object module files. **tld** (Trip Link eDitor) links object module files, generating absolute code.

For each application program, a block of ROM was reserved for its code and data constants. The size of the reserved block was the smallest power of two that was greater than the required size of the specific application program. The unused portion of the reserved block was filled with the byte value zero. As can be concluded from the sizes of the reserved blocks, as shown in Table 1, a large portion of the 1 MByte ROM was unused. This portion was filled with repeated copies of a small dummy program with a code size of 72 bytes and a reserved block size of 128 bytes. The unused portion of the reserved block was filled with the byte value zero. The dummy program consisted of an endless loop with a mean basic block size of 4 instructions. The RAM was initialized with zeros.

Program Characteristics	PILOT	HSORT
Proportion of watchdog instructions	13%	25%
Mean basic block size [instructions]	9.4	4.8
Code Size [Bytes]	1904	844
Size of Reserved Block in ROM [KBytes]	4	2

Table 1 Application Program Characteristics.

It should be noticed that the range of the mean basic block sizes of 4.8 to 9.4 for the application programs, as shown in Table 1, spans the range of typical mean basic block sizes of 4 to 10 instructions suggested by other studies [10], [14]. The mean basic block size of 4 for the dummy program was deliberately a worst-case choice, which increased the risk of undetected control flow errors resulting from the special case in which the signature was zero and an erroneous branch was made to program space occupied by the dummy program. In that case, the probability was 0.20 that the branch destination would be a branch-in point and, consequently, the control flow error would be undetected.

4.2 Error Outcome and Classification

A single **flip-to-1** (ft-1) error is defined according to Karlsson *et al.* [7] as the manifestation of a fault in one information bit such that the logical value is changed from zero to one. A single **flip-to-0** (ft-0) error is defined in analogy with a single ft-1 error. The fault injection method of toggling a single bit in an internal state element always resulted in the manifestation of the fault as an error in one information bit, and thus every fault injection attempt resulted in either a ft-0 or a ft-1 error. Both experiments

resulted in a significantly larger proportion of ft-1 (77%) than ft-0 (23%) errors, indicating that internal registers more frequently contained the bit value zero than one.

The outcome of each individual fault injection attempt was classified into one of the following three main groups: effective errors, overwritten errors and errors still latent after 100 clock cycles. The following error classification is used:

An error that propagates to any output pin of TRIP is called an **effective** error except when it only affects the status pins, WD_ERR (indicates the detection of an error by the watchdog) or PAR_ERR (indicates the detection of an error by the parity-checking mechanism). An error is termed **overwritten** when the internal state of TRIP no longer differs between the reference and the fault injection simulations and only when the error has not previously been classified as effective. Errors that have neither become effective nor overwritten at the end of a simulation are called **latent**.

Only 23% of all injected faults resulted in effective errors, while 34% became overwritten and the remaining 43% became latent. Latent errors were divided into two classes: latent errors in used and unused user registers. Latent errors in unused registers were the result of 24% of all injected faults, while 18% resulted in latent errors in used registers. Latent errors in unused registers will never become effective, while latent errors in used registers will probably eventually become either effective or overwritten. The HSORT experiment resulted in a greater proportion of latent errors in unused registers than the PILOT experiment, owing to the fact that the number of unused general purpose registers were 15 in HSORT and only 9 in PILOT. The PILOT and HSORT experiments resulted in similar proportions of effective errors (23% and 25%) and overwritten errors (32% and 35%). A program dependency was observed for the proportions of latent errors in used registers. The proportion of these errors was greater for the PILOT experiment (22%) than for HSORT (11%). The outcome of both experiments is summarized in Table 2.

Outcome of Experiment			PILOT	HSORT	Total
Fault Injection Attempts			1849	930	2779
Error Type	ft-1		1392 (75.3%)	744 (80.0%)	2136 (76.9%)
	ft-0		457 (24.7%)	186 (20.0%)	643 (23.1%)
Error Outcome	Effective Errors		416 (22.5%)	230 (24.7%)	646 (23.2%)
	Overwritten Errors		650 (35.2%)	297 (31.9%)	947 (34.1%)
	Errors still latent after 100 cycles	in unused registers	380 (20.6%)	297 (31.9%)	677 (24.4%)
		in used registers	403 (21.8%)	106 (11.4%)	509 (18.3%)

Table 2 Outcome of Fault Injection Experiments.

All fault injection attempts that resulted in effective errors were classified into one of three groups: control flow error, data error or other error. The following effective error classification is used:

An error is termed a **control flow** error if it causes a change in the instruction flow which does not correspond to any of the alternative paths provided by the previous instruction. Special events such as traps, interrupts and exceptions are not classified as errors unless they are caused by a fault injection. However, if a trap or interrupt is caused by a fault injection affecting the surprise-handling mechanism, it is classified as a control flow error. When an exception induced by a fault injection is detected, the cause of the exception determines the classification of the error. If a bus error is caused by an instruction fetch or if an address error is caused by a branch to an illegal address or if an illegal opcode is fetched, the error is classified as a control flow error. Privilege violation exceptions caused by fault injections are always classified as control flow errors. An error that causes a load or store instruction to reference an incorrect address, or causes a store instruction to write incorrect data, is called a **data** error. Exception causes are not normally classified as data errors, except in two cases: when a bus error or an address error is caused by the memory access of a load or store instruction. An error that has not been classified as either control flow or data error at the end of a simulation is given the term of **other** error.

An error classified as a control flow error cannot later also be classified as a data error, while an error first classified as a data error can also later be classified as a control flow error. Examples of other errors are conditional branches which take an incorrect but legal branch path owing to an erroneous branch condition and errors causing TRIP to enter any one of the following internal states: halt, reset or fail.

The results show that 33% of all effective errors resulted in control flow errors, while 60% resulted in data error. The remaining 7% were classified as other errors, as shown in Table 3.

Error Class	PILOT	HSORT	Total
Control Flow	133 (32.0%)	80 (34.8%)	213 (33.0%)
Data	263 (63.2%)	124 (53.9%)	387 (59.9%)
Other	20 (4.8%)	26 (11.3%)	46 (7.1%)
Total	**416 (100%)**	**230 (100%)**	**646 (100%)**

Table 3 Classification of Effective Errors.

4.3 Performance Overhead

The **performance overhead** is defined as the fraction of the total execution time the processor spends executing watchdog instructions. A performance overhead of 0% indicates that no watchdog instructions are executed, and

an overhead of 100% indicates that nothing but watchdog instructions are executed. As described previously in the section *Description of TRIP*, **watchdog instructions** are divided into two groups: `tst` instructions and watchdog-supporting instructions.

Table 4 shows that the performance overhead varied notably with the workload, from 9% for PILOT to 19% for HSORT. The performance overhead resulting from the execution of watchdog-supporting instructions was approximately 4% for both application programs, while the overhead resulting from the execution of `tst` instructions varied between 5% for PILOT and 15% for HSORT.

Instruction Class	PILOT [cycle]	HSORT [cycle]
Application	18208 (91.2%)	8191 (81.0%)
tst	1057 (5.3%)	1493 (14.8%)
Watchdog-Supporting	694 (3.5%)	423 (4.2%)
Watchdog Total	1751 (8.8%)	1916 (19.0%)
Total	19959 (100.0%)	10107 (100.0%)

Table 4 Execution Statistics for PILOT and HSORT.

Watchdog-supporting instructions were inserted at the beginning and end of each subroutine for the primary purpose of saving/retrieving the instruction counter to/from the stack. Thus, the performance overhead resulting from the execution of watchdog-supporting instructions was dependent upon how often subroutine calls were executed in relation to other instructions.

The performance overhead resulting from the execution of `tst` instructions depends on the dynamic mean basic block size and on the instruction mix of the basic blocks executed. Both application programs show a coarse correspondence between their dynamic and static mean basic block sizes. The dynamic mean basic block size was 4.3 instructions for HSORT and 12.5 for PILOT, while the static mean basic block size was 4.8 and 9.4, respectively, for HSORT and PILOT.

Notice that the performance overhead obtained is pessimistic, as it is assumed that a `tst` instruction always represents an execution overhead. Normally this is not true, as it is not always possible to fill the delay slot with a meaningful instruction.

4.4 Coverage Evaluation

The **error detection coverage** is defined as the proportion of effective errors that are detected.

It was observed that the total error detection coverage was low and was also workload-dependent; only 45% were detected for PILOT as compared with 60% for HSORT, as shown in Table 5. A slight workload dependency was observed for all error detection methods for their coverage of control flow errors, while the workload dependency of the coverage of data errors was notable for both the bus and address error mechanisms.

Error Class	PILOT					HSORT				
	BE	AE	BE + AE	BAS	WD + BAS	BE	AE	BE + AE	BAS	WD + BAS
Control Flow	50 37.6%	13 9.8%	63 47.4%	67 50.4%	131 98.5%	31 38.8%	3 3.8%	34 42.5%	37 46.2%	76 95.0%
Data	18 6.8%	9 3.4%	27 10.3%	27 10.3%	58 22.1%	33 26.6%	16 12.9%	49 39.5%	49 39.5%	62 50.0%
Other	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 3.8%
Total	68 16.3%	22 5.3%	90 21.6%	94 22.6%	189 45.4%	64 27.8%	19 8.3%	83 36.1%	86 37.4%	139 60.4%

Table 5 Error Detection Coverage.

Independent of workload, the basic error-detecting mechanisms detected 48% of all control flow errors, while the error detection coverage of data errors was clearly workload-dependent; only 10% of all data errors were detected by the basic error-detecting mechanisms for PILOT, while 40% were detected for HSORT. The privilege violation and illegal opcode mechanisms only detected between 3 and 4 control flow errors that the bus and address error mechanisms did not, and they detected no extra data errors. The reason that the illegal opcode mechanism detected so few control flow errors was that the bit-pattern 0...0 was a legal instruction.

The inclusion of the watchdog mechanism raises the total detection coverage of control flow errors to 98.5% for PILOT and to 95.0% for HSORT. It should be noted that an exception handler was executed immediately when a basic error-detecting mechanism detected an error, which made it impossible to evaluate the error detection coverage of the watchdog mechanism alone, as it never received the opportunity to detect those errors. The instruction counter timeout mechanism contributed to one-third of the total error detection coverage of the watchdog.

By manual inspection of the simulation data, it was observed that erroneous traps, i.e. traps caused by a fault affecting the surprise-handling mechanism, caused 11 (8.3%) of the control flow errors for PILOT and 8 (10%) for HSORT. The watchdog mechanism detected 10 of the 11 errors for PILOT and 6 of the 8 errors for HSORT, resulting in a detection coverage of 75% and 90%, respectively, for erroneous traps. An erroneous trap was undetected when an error caused a trap just after a `tst` instruction had been executed, i.e. when the signature register had been cleared. Thus, erroneous traps were undetected because of correlation between initial intermediate signatures.

The remaining control flow errors that were undetected by the watchdog mechanism (2 for PILOT and 1 for HSORT) were also so because of the correlation between

initial intermediate signatures, as described by Wilken *et al.* [21]. If the erroneous traps are neglected, the total detection coverage of control flow errors becomes 98.6% for HSORT and 99.1% for PILOT, which can be compared with the coverage predicted by Wilken *et al.* [21] of 97.5% and 99.5%.

It was observed that the bus error mechanism detected more control flow errors than the address error mechanism, which was expected, as the bus error mechanism can detect illegal instruction fetches from unmapped memory, while the address error mechanism can only detect errors which cause the destination address of a branch to be unaligned, which will happen only if one of the two low-bits of the address are affected.

The error detection coverage of data errors for both the bus error and the address error mechanisms was strongly workload-dependent. This can be explained by the difference in how memory access addresses are computed. As described in section *Error Outcome and Classification*, the cause of a bus or address error exception was only classified as a data error if the exception was caused by the memory access of a load or store instruction. The memory access addresses of the HSORT program are computed in a more complex way than in the PILOT program, as the former sorts an array stored in memory. This means that the probability of multiple-bit errors in the memory access address is high, which increases the probability of detection by the bus and address error mechanisms. PILOT, on the other hand, mainly stores results of arithmetic computations at fixed addresses. Thus the probability of multiple-bit errors affecting the memory access address is low. However, independent of workload, the bus error mechanism detected twice the number of errors than did the address error mechanism. It can be concluded that the bus error mechanism was superior to the address error mechanism at detecting both control flow and data errors.

No data errors were detected by the privilege violation or illegal opcode mechanisms.

The watchdog added 12% to the detection coverage of data errors for PILOT and 10% for HSORT. There were primarily two cases in which the watchdog detected data errors: when the frame pointer was affected by the fault and when the fault was injected into one of the instruction registers (INS_{OF} and INS_{EX} in Figure 2). A frame pointer error caused the instruction flow to diverge because an illegal return address was popped from the memory area pointed to by the frame pointer. This was detected by the watchdog, as it resulted in an incorrect branch. The error manifested itself as a data error when the return address was fetched. An error in one of the instruction registers could cause a data computation to yield an incorrect result if an arithmetic instruction was affected. The watchdog would detect the incorrect instruction when the signature was checked, as the signature is based on the bit-pattern of the instruction register. However, a control flow error would not be observed on the processor pins, as the correct instruction was originally fetched from memory. The error will later be manifested as a data error when the incorrect register value is stored in memory or used as an address register in a bus access.

Only one error classified as 'other' error was detected.

4.5 Latency Evaluation

The **error detection latency** is defined as the number of cycles that passes from the time the error is manifested on the pins, i.e. it becomes effective, to the time when the error is detected.

The error detection coverage of control flow errors is plotted as a function of the detection latency in Figure 3 for PILOT and for HSORT. The frequency with which checked blocks with different execution times were executed is also plotted in Figures 3. The frequency is divided by the total number of executed checked blocks for each application program.

The address error mechanism detected a majority of its control flow errors with a detection latency shorter than 5 clock cycles. For the PILOT program, it also detected a few errors with a longer detection latency of about 35 clock cycles. These errors were injected when TRIP was executing one of the longer checked blocks that had an execution time of 53 and 135 clock cycles, respectively.

The bus error mechanism, just as the address error mechanism, detected the majority of its control flow errors with a detection latency less than or equal to zero clock cycles. The remaining detected control flow errors were detected with a latency that appears to be correlated to the execution time of the most frequently executed checked blocks. Such a correlation can be explained by the fact that checked blocks often contain a branch instruction as the next to last instruction. That instruction could be a 'return from subroutine' instruction with the return address popped from the stack or an unconditional branch instruction with the destination address stored in a register. An error in the stack or frame pointer in the first case, or in the destination register in the second case, could cause an illegal branch to unmapped memory, which will cause a bus error.

For the watchdog mechanism, it can be expected that the error detection latency for control flow errors is correlated to the execution time of the most frequently executed checked blocks, as errors would be detected only at the end of the checked block when the `tst` instruction is executed. As expected, the latency distribution increases more rapidly around points in time corresponding to one-half of the execution time of the most frequently executed checked blocks.

The median error detection latency was -1 clock cycle for the address error mechanism, as shown in Table 6, as most errors were detected internally in TRIP before they propagated to the pins.

The median error detection latency was 0 clock cycles for the bus error mechanism, as a bus error signal was instantly generated by external logic when the error was manifested on the processor pins. The error detection latency for the basic error-detecting mechanisms was dominated by the influence of the bus error mechanism owing to the fact that the bus error mechanism detected more errors than the address error mechanism. The difference in error detection latency was small for the basic error-detecting mechanisms as compared with the combined latency for the bus and address error mechanisms.

Error Class		PILOT					HSORT				
		BE	AE	BE + AE	BAS	WD + BAS	BE	AE	BE + AE	BAS	WD + BAS
Control Flow	Median	0	-1	0	0	1	0	-1	0	0	1
	Mean	0.1	5.2	1.2	1.0	9.8	2.3	1.7	2.2	1.9	2.7
Data	Median	0	-1	0	0	4.5	0	-1	0	0	0
	Mean	2.8	1.9	2.5	2.5	13.3	1.3	9.4	4.0	4.0	1.4
Other	Median	-	-	-	-	-	-	-	-	-	1
	Mean	-	-	-	-	-	-	-	-	-	1
Total	Median	0	-1	0	0	2	0	-1	0	0	0
	Mean	0.8	3.8	1.6	1.4	10.9	1.8	8.2	3.3	3.1	2.1

Table 6 Error Detection Latency [cycle].

The median error detection latency for the combined watchdog and basic error-detecting mechanisms was 1 clock cycle for control flow errors, while a workload dependency of the mean latency was observed. The mean latency was 10 cycles for PILOT and 3 for HSORT, which can be explained by the large mean dynamic checked block execution time of 18.9 cycles for PILOT and the shorter execution time of 6.8 cycles for HSORT. As expected, these latency figures correspond to one-half of the average execution time of a checked block.

Both the mean and median error detection latency of data errors for the combined watchdog and basic error-detecting mechanisms was workload-dependent. The latency was on average larger for the PILOT program than for HSORT, depending on the difference in their mean dynamic checked block sizes.

Some data errors can be expected to generate control flow errors after the observed simulation time. Examples are data errors affecting the frame pointer. It can be predicted that, for several of these errors, it will take in the order of 1000 clock cycles until the error causes a control flow error.

To summarize, the mean error detection latency was short (<4 clock cycles) for the basic error detection mechanisms and only slightly longer when the effect of the watchdog was considered(<14 clock cycles).

5 Summary and Conclusions

In this paper, ESM was adapted for a pipelined RISC architecture. It was concluded that the use of BAH or JS were either not appropriate or did not improve performance if an upper boundary for the error detection latency for control flow errors must be guaranteed. A basic ESM technique was chosen for this reason. A special machine instruction was added to the instruction set of the RISC for signature comparison and initialization.

A total of 2,779 simulated fault injections were performed. The results of these simulations showed that only 33% of all errors that became effective resulted in control flow errors, while 60% resulted in data errors. This can be compared to the figures of 77% for control flow errors and 18% for data errors obtained for physical fault injections performed by Gunneflo *et al.* [6] in an 8-bit non-pipelined CISC (MC6809E). The difference in error behavior is

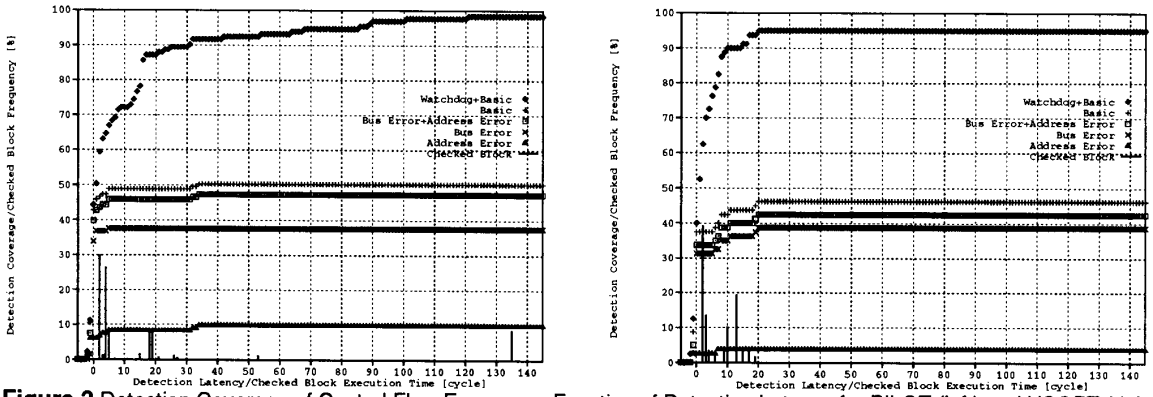

Figure 3 Detection Coverage of Control Flow Errors as a Function of Detection Latency for PILOT (left) and HSORT (right) Compared with the Frequency of Different Checked Block Execution Times.

most likely the result of architectural variations between RISC and CISC processors. The high proportion of data errors indicates that efficient data error-detecting methods are more important for RISC than for CISC processors.

To summarize, the results indicate that the proposed modified ESM method with low chip area overhead (4.7%) can be added to a processor, and provides high error detection coverage of control flow errors (between 95.0% and 98.5%) and a predictable and low (mean lower than 10 clock cycles) error detection latency. The performance overhead was workload-dependent, and varied between 9% and 19%. Furthermore, the experimental results show that the processor exceptions, bus error and address error, contribute to the rapid detection of more than 40% of the control flow errors. The bus error mechanism had a higher coverage than address error for both data and control flow errors.

As faults were only injected into registers, it was not possible to evaluate the efficiency of the parity-checking method, and thus it was impossible to determine a total error detection coverage figure. If the utilization of a single error-detecting processor is to be competitive with duplication and comparison, it is necessary to detect a very high proportion of both data and control flow errors. For instance, an effort should be made to improve the efficiency of the watchdog method only if it limits the total error detection coverage. As all undetected control flow errors were caused by the correlation between initial intermediate signatures, it should be possible to improve the watchdog method by reducing this correlation. Simulation experiments are planned that will address this problem.

References

[1]A. Carlsson, "RISC EVALUATION STUDY; WORK PACKAGE 4 - Ada BENCHMARK RUNNING", Saab Space AB, Göteborg, Sweden, 1990.

[2]X. Delord, G. Saucier, "Control flow checking in pipelined RISC microprocessors: the Motorola MC88100 case study", EUROMICRO'90 Workshop on real time, 1990, pp. 162-169.

[3]J. B. Eifert, J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams", FTCS-14, IEEE, 1984, pp. 394-399.

[4]I. D.Elliot, I. L. Sayers, "Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction", IEE PROCEEDINGS, Vol. 137, Pt. E, No. 1, Jan 1990, pp. 88-102.

[5]R. Firth, "Core Set of Assembly Language Instructions for MIPS-based Microprocessors - Draft Version 3.2", Software Engineering Institute, Pittsburgh, USA.

[6]U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", FTCS-19, IEEE, 1989, pp. 340-347.

[7]J. Karlsson, P. Lidén, "Transient Fault Effects in the MC6809E 8-bit Microprocessor: A Comparison of Results of Physical and Simulated Fault Injection Experiments", Technical Report No. 96, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1990.

[8]R. Leveugle, T. Michel, G. Saucier, "Design of Microprocessors with Built-In On-Line Test", FTCS-20, IEEE, 1990, pp. 450-456.

[9]D. J. Lu, "Watchdog processor and structural integrity testing", IEEE Trans. Comput., vol. C-31, July 1982, pp. 681-685.

[10]A. Mahmood, E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processor - A Survey", IEEE Trans. Comp., VOL. 37, NO. 2, Feb. 1988, pp. 160-174.

[11]M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation", ITC, IEEE, 1982, pp. 461-468.

[12]M. Namjoo, "CERBERUS-16: an architecture for a general purpose watchdog processor", FTCS-13, IEEE, 1983, pp. 216-219.

[13]M. Nicolaidis, "Evaluation of a Self-Checking Version of the MC 68000 Microprocessor", Microprocessing and Microprogramming, No. 20, 1987, pp. 235-247.

[14]M. A. Schuette, J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams", IEEE Trans. Comp., Vol. C-36, NO. 3, March 1987, pp. 264-276.

[15]J. P. Shen, M. A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams", ITC, IEEE, 1983, pp. 275-282.

[16]D. P. Siewiorek, V.R. S. Swarz, The Theory and Practice of Reliable System Design. Bedford, MA: digital, 1982, ch. 3.

[17]J. Sosnowski, "Detection of control flow errors using signature and checking instructions", ITC, IEEE, 1988, pp. 81-88.

[18]T. Sridhar, S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors", ITC, IEEE, 1982, pp. 191-199.

[19]S. P. Tomas, J. P. Shen, "A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems", Int. Conf. On Computer Design, 1985, pp. 531-539.

[20]J. Torin, O. Bridal, M. Rimén, R. Snedsbøl, "Architecture of a Dependable Computer Network for Control of Vehicle Dynamics", Proc. 4th PROMETHEUS Workshop, 1990, pp. 164-173.

[21]K. Wilken, J. P. Shen, "Embedded Signature Monitoring: Analysis and Technique", ITC, IEEE, 1987, pp. 324-333.

[22]K. Wilken, J. P. Shen, "CONTINOUS SIGNATURE MONITORING: Efficient Concurrent-Detection of Processor Control Errors", ITC, IEEE, 1988, pp. 914-925.