Software Modification Aided Transient Error Tolerance for Embedded Systems

Rishad A Shafik[†], Gerard Rauwerda[‡], Jordy Potman[‡], Kim Sunesen[‡], Dhiraj Pradhan[†], Jimson Mathew[†], Ioannis Sourdis[§]
[†]*University of Bristol, UK;* [‡]*Recore Systems, Enschede, Netherlands;* [§]*Chalmers University of Technology, Sweden*
E-mail: [†]*ras06r@zepler.org,* [‡]*gerard.rauwerda@recoresystems.com,* [§]*sourdis@chalmers.se*

***Abstract*—Commercial off-the-shelf (COTS) components are increasingly being employed in embedded systems due to their high performance at low cost. With emerging reliability requirements, design of these components using traditional hardware redundancy incur large overheads, time-demanding re-design and validation. To reduce the design time with shorter time-to-market requirements, software-only reliable design techniques can provide with an effective and low-cost alternative. This paper presents a novel, architecture-independent software modification tool, SMART (Software Modification Aided transient eRror Tolerance) for effective error detection and tolerance. To detect transient errors in processor datapath, control flow and memory at reasonable system overheads, the tool incorporates selective and non-intrusive data duplication and dynamic signature comparison. Also, to mitigate the impact of the detected errors, it facilitates further software modification implementing software-based check-pointing. Due to automatic software based source-to-source modification tailored to a given reliability requirement, the tool requires no re-design effort, hardware- or compiler-level intervention. We evaluate the effectiveness of the tool using a Xentium® processor based system as a case study of COTS based systems. Using various benchmark applications with single-event upset (SEUs) based error model, we show that up to 91% of the errors can be detected or masked with reasonable performance, energy and memory footprint overheads.**

***Keywords*-Fault Tolerance, Error Detection, Reliable Computing, Embedded Systems**

I. Introduction

Continued technology scaling has enabled the fabrication of ever more efficient and low power electronic devices for current and future generation of embedded systems. Examples of such devices include IBM's 22-nm [5] and Intel's emerging 16-nm [17] devices with promises to provide with unprecedented integration capacity and performance. However, with these technological advances, design complexity is also increasing significantly with emerging challenges. A major challenge for such systems design is the increasing number of transient errors caused by cross-talk, power supply noise and neutron or alpha radiation [9]. These errors manifest themselves as temporary logic upsets, such as single-event upsets (SEUs), and can affect the signal transfers and stored values leading to incorrect execution in embedded systems. Operating reliably in the presence of these errors is a crucial requirement for many applications, particularly for high availability, safety-critical systems [7].

To mitigate the impact of these errors, designers typically have to address two related issues: error detection, followed by error correction or mitigation. Error detection deals with identifying the presence of one or more errors in the system, which are then suitably corrected or mitigated using various design approaches. Traditionally reliable design with error detection and tolerance capabilities is carried out through various approaches, including hardware redundancy [8], time/information redundancy [4] or radiation hardening [9]. However, these approaches incur large performance, resource and energy overheads. Moreover, meeting shorter time-to-market requirements can be highly challenging as re-design and re-validation are required [1].

Recently, to meet emerging reliability requirements of current and future embedded systems at low cost, commercial vendors are increasingly employing commercial off-the-shelf (COTS) components, particularly in avionic control, implantable/wearable medical and signal processing devices [2]. These components have little or no radiation hardening or hardware customizations and provide with high performance at low cost [18]. Over the years, researchers have proposed various other design approaches to incorporate error tolerance for COTS based systems. For example, to achieve reliability with low hardware overheads Rashid *et al.* proposed a time redundancy approach using instruction-level re-execution [13]. Using instruction-level parallelism at compiler-level the performance overheads can be reduced in this approach. A similar approach using instruction-level duplication technique for error detection in COTS DSP processors using multiple execution units has been proposed by Bernardi *et al.* [1]. The duplicated executions in [1] are generated by hardware support units in the processor, which are then compared in a separate unit to detect errors. Pignol proposed another approach using task-level redundancy for error detection and tolerance in DMT and DT2 architectures [12]. In this approach, error detection is achieved through re-execution of the computation tasks using a memory bridge, followed by comparisons of the results. Among others, system-level approaches to re-execution has also been demonstrated in [18] by Troxel *et al*. Their approach highlighted the effectiveness of error mitigation technique using application check-pointing and roll-back in COTS based systems [18]. To facilitate such check-pointing, a middleware design was proposed in a COTS based DM system architecture [18].

Hardware and time redundancy approaches require direct hardware modification in the original system (such as, comparator design [1] or incorporating watchdog timers [13]). However, in many COTS based systems, adding or modifying hardware resources may not be feasible due to time-to-market and low design cost is-

sues. For such systems, an alternative is to use compiler based optimizations or modification to introduce software implemented error tolerance. An example of this is instruction duplication technique (called EDDI), proposed by Oh *et al.* [11]. The instruction duplication in [11] is facilitated by compiler-generated different registers and memory locations for each set of duplicated instructions. The outputs of these instructions are then compared to detect errors. Reis *et al.* further enhanced EDDI in their software implemented fault tolerance (SWIFT) technique by taking out memory replications with an assumption of error correction coding enabled memory unit [16]. As a result, SWIFT achieved lower overall system overheads with reasonable error detection capabilities. A similar approach with compiler-assisted data duplication was proposed by Hu *et al.* in [6]. In their approach the compiler determines the instruction schedule by trading off between the performance degradation caused by the degree of duplication and the achievable reliability.

The above techniques require designers' direct intervention in hardware design [1], [12] or software compilers [10], [16] to incorporate desired error detection and tolerance capabilities. However, in many COTS based systems, this may not be feasible due to intellectual property (IP) rights and cost control. Hence, software-only solutions can be highly coveted due to two factors. Firstly, no time-demanding re-design and validations are required. Secondly, it comes free of cost as it requires no designers' intervention in hardware or software compilers [15]. In this paper, we propose a software-only and architecture-independent software modification technique implemented in a *novel* prototype tool, called SMART. We show that the proposed technique can automatically incorporate software modification for effective error detection through data duplication and signature comparison, and correction through software implemented check-pointing. We will demonstrate that with reliability tailored modification the tool can achieve high transient error detection coverage and effective error tolerance at reasonable performance and memory footprint overheads.

The rest of the paper is organized as follows. Section II gives the preliminary details including system architecture and error injection mechanism, while Section III presents the proposed SMART model using software modification based error detection and tolerance. Section IV demonstrates the effectiveness of the proposed technique using a number of benchmark applications, highlighting the comparative advantages and related system overheads. Finally, Section V concludes the paper.

II. Preliminaries

System architecture and error injection environment used in this work are briefly described.

A. System Architecture

Figure 1 shows the system architecture used in this work. The system architecture includes a high-performance fixed-point Xentium® DSP processor [19], used as a case study of COTS components. The processor consists of the Xentium® core containing the datapath, control and instruction cache. The datapath contains ten arithmetic and logic execution units, supporting instruction-level parallelism (ILP) for 32/40-bit scalar and two 16-bit element vector operations. The Xentium core is integrated with a tightly coupled data memory (TCM) for storage, which can be further extended through data interface connection to an on-chip data memory unit. The sizes of the instruction cache, TCM and data memory are configurable. Xentium®

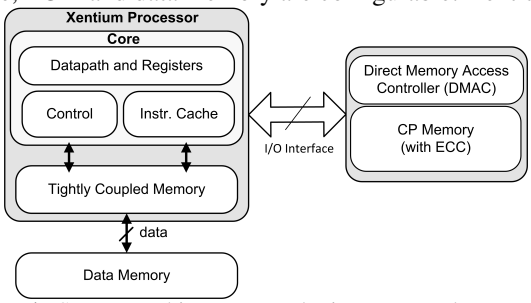

Figure 1: System architecture employing proposed error detection and tolerance technique

processor also incorporates an IO interface for enabling interconnections and on-chip communications. In this work, Xentium® without hardware error detection capability and watchdog timer support is considered. To enable storage of application check-points, the IO port is connected to an external memory component, interfaced via a direct memory access controller (DMAC) unit. The memory is assumed to be sufficiently coded with error detection and correction (EDAC) codes to facilitate error-free assumption during check-point saving and retrieval. Similar memory has also used in [16].

B. Error Injection

Transient errors manifest themselves as temporary logic upsets at circuit-level. However, at application-level the impact of these errors depends on the instructions being executed or datapath/memory activity. For example, if a transient error takes place during loading of a memory word into the processor datapath, it can leave the values being incorrectly processed by that word. Similarly, an error in the datapath can also render wrong instruction being executed. To effectively reflect these and various other manifestations at instruction level, error injections are carried out through instruction perturbation through Instruction Set Simulator (ISS) using a separate parser, as shown in Figure 2(a). Single-event upset (SEU) based model is used for error injection, similar to [16].

As can be seen, error injection is enabled by instruction-level (assembly) parser, which adds the original application instructions. First, a register flag variable $fault$ is incorporated, which is set to $true$ to enable fault injections or vice versa. When this flag is set, the impact of error is modeled through any of the following: errors in datapath (through change of instruction, replacing the operand values or the actual operands), errors in program control flow (through change of jump label or storage registers), errors in mem-

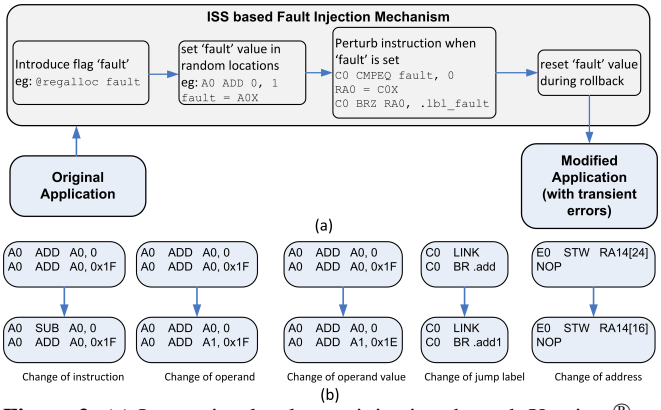

Figure 2: (a) Instruction-level error injection through Xentium® ISS, (b) Examples of error injections

ory (through change of value or offset address during loading from or storing into memory). Figure 2(b) shows few examples of such instruction-level perturbations. To ensure the errors are cleared during error injection simulation, the $fault$ variable is reset in the roll-back routine (SMART-CP, Section III-C). To control timing and random location of the SEU injection, the original application is first profiled using the cycle-accurate output trace from the ISS. The application profiling gives statistics of different instruction types for a given application, leading to relative error probabilities for a given fault rate. Section IV further reports a number of error injection experiments for different applications.

III. Proposed SMART Model

Figure 3 shows the proposed Software Modification Aided transient eRror Tolerance (SMART) prototype tool organized in three modification steps: high-level modification to incorporate transient error detection (called SMART-ED), followed by assembly-level modification to detect illegal branching (called SMART-SC), and finally assembly-level modification to mitigate the impact of detected errors through application check-pointing routines (called SMART-CP). Details of SMART steps follows.

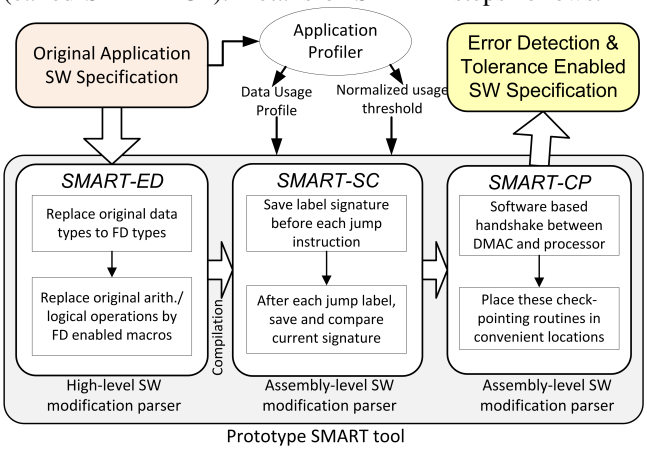

Figure 3: Prototype SMART tool for software implemented error detection and tolerance

A. SMART-ED: Error Detection through Duplication

Error detection in SMART (SMART-ED) is initiated by using a high-level parser, which replaces the original data types to error detection enabler types in the modified software description (Figure 3). The replacement data types, which are defined in a separate pre-compiled library, automatically generate duplicate copies of the original data. To provide with original functionality with error detection capability and also to detect errors during computation, the high-level arithmetic and logical operations are also replaced by equivalent macro computation functions. These function, defined in the same pre-compiled library, compare the duplicate copies during arithmetic or logical operations to detect transient errors.

To illustrate the software modification in SMART-ED, Figure 4(a) and (b) show example original and modified descriptions in C. As can be seen, the original software de-

```
1: //Include Libraries
   ............
2: int a;
3: long b;
4: long c[10];
5: ......
6: a = 34;
7: b = 29;
8: c[3] = (long) a * b;
   ............
```

(a)

```
1: //Include Libraries
2: #include<modify.h>
   ............
3: FDInt a;
4: FDLOng b;
5: FDLOng c[10];
6: ......
7: AssignVal(a, 34);
8: AssignVal(b, 29);
9: AssignVar(ArrayElement(c,
     3), MultiplyVar(a, b));
   ............
```

(b)

Figure 4: Example C statements of (a) original software description, and (b) modified description generated in SMART-ED

scription undergoes two major changes. First, the original variable a, b and c declarations with types int and $long$ in lines 2-4 are modified to $FDInt$ and $FDLong$ types in lines 3-5 (Figure 4(b)). Then to carry out the original functionality in lines 6-7 (Figure 4(a)) using the modified types for a and b, $AssignVal$ macro function is used in the modified software description in lines 7-8 (Figure 4(b)). The $AssignVal$ function carries out the assignment of the variables but also carries out the comparisons between their respective copies after the assignment operation. Similarly, for assigning the resulting value of $(a * b)$ in an array element of c (line 8, Figure 4(a)), $AssignVar$ macro function is used to assign variable value generated by $MultiplyVar$ in array element of c using $ArrayElement$ macro function in the modified description (line 9, Figure 4(b)).

To illustrate how these type replacements and functional modifications (in Figure 4(b)) enable transient error detection, Figure 5 shows definitions of type $FDInt$ (in (a)) and macro function $MultiplyVar$ (in (b)) as examples. As can be seen, the definition of type $FDInt$ in the modification library $modify.h$ generates duplicate copies of the data with the same original int type (lines 3-4, Figure 5(a)). These duplicate copies (i.e. $orig$ and $copy$) are compared during the computation (i.e. multiply operation defined in *modify.h*) to detect the presence of one or more errors in SMART-ED. Note that the comparisons are carried out in two instances. In the first instance, comparison is carried out to detect the presence of errors (through checking any difference between the duplicate copies) incurred in memory or during loading into registers

```
// defined in modify.h
1: unsigned short detected = 0;
2: typedef struct _smartedInt{
3:       int orig;
4:       int copy;
5: } FDInt;
```

(a)

```
// defined in modify.h
1: #define MultiplyVar(op1, op2) ({
2: if(!detected=(op1.orig!=op1.copy||op2.orig!=op2.copy))
     {
3:    op1.orig = op1.orig * op2.orig;
4:    op1.copy = op1.copy * op2.copy; }
5: else {sw_roll-back();}
6: if(detected=(op1.orig!=op1.copy||op2.orig!=op2.copy))
7:    {sw_roll-back();}
8: })
```

(b)

Figure 5: (a) Example error detection enabler type $FDInt$ (replacement of int data type), and (b) Macro function $MultiplyVar$ replacing $*$ operation

(line 2, Figure 5(b)). If any error is detected (notified by $detected$ variable, defined in the $modify.h$ library), further computation is avoided and software implemented roll-back routine is invoked (line 5, further details of software implemented check-pointing and roll-back can be found in SMART-CP, Section III-C). If no error is detected in the first comparison, computation (i.e. multiplication operation) is carried out for the duplicate copies (lines 3-4). This is then followed by the second comparison to detect any error incurred in the datapath or its registers after the actual computation (line 6). If any error is detected in this instance, software implemented roll-back routine ($sw\_roll-back$) is invoked (line 7).

The comparisons in two instances have an added advantage apart from saving duplicate computations when one or more errors are detected (lines 2 and 6). When $detected$ variable itself is subjected to errors during the first comparison, following the computation can reset $detected$ variable and trigger $sw\_roll-back$ an error is detected. However, if $detected$ variable is subjected to errors during the second comparison (line 6), incorrect computations can take place.

To control the potential performance overhead due to duplication of data and their computations, SMART incorporates two different inputs generated by the application profiler (Figure 3): data usage profile and normalized usage threshold (U_{NT}). The data usage profile gives the structured organization of the various variables and constants used in an application, together with their usage statistics. Using such data usage statistics, the tool only duplicates the original data types and operations of chosen variables or constants that have usage higher than normalized threshold (i.e. $U_N \geq U_{NT}$). Normalized threshold usage (U_{NT}) is calculated as a ratio of a variable's usage count to the maximum usage of any other variable. Details of how data/register usage is generated can be found in [14].

B. SMART-SC: Dynamic Signature Comparison

Branch instructions are vulnerable components that control the software flow [16]. Transient errors can affect the software flow and often cause software to terminate abruptly or continue execution with unintended functionality. Hence, to detect such illegal branch instructions dynamic signature comparison is carried out with assembly-level software modification (Figure 3). SMART-SC procedure is as follows:

(a) The original branch instruction is preceded by instructions to save the function of target label address (L_{src}).
(b) After each label similar function of current instruction packet address is saved (L_{dst}).
(c) L_{src} and L_{dst} are then XORed to generate a signature value ($S = L_{src} \oplus L_{dst}$).
(d) If S contains unexpected value, it is considered an illegal branch.

To incorporate signature saving and checking mechanism in an architecture-independent manner using the above procedure, SMART takes branch instruction syntax for various branch instructions as input (not shown in Figure 3). Hence, the software goes through further modifications to detect illegal control flows due to transient errors. Figure 6 shows an example demonstration of these procedures to detect illegal branch instructions. For illustration purposes, partial original and modified Xentium® assembly descriptions are shown in Figures 6(a) and (b). As can be seen, the actual branch instruction takes place

```
1:    NOP
2:    C0 BR .add
3:    NOP 2
      ;......
4: .add:
5:    A0 ADD RA0, RB0
6:    RA0 = A0X
      ;.....
7:    C0 BR .multiply
8:    NOP 2
      ;.....
9: .multiply:
10:   M0 MUL RA0, RB0
11:   NOP
      ;.....
```

(a)

```
1:    NOP
2:    A0 ADD 0, lo16(.add)
3:    A0 ADDU A0X, hi16(.add)
4:    RA8 = A0X
5:    C0 BR .add
6:    NOP 2
7: .add:
8:    C0 LINK
9:    C0 XOR C0X, RA8
10:   RA1 = C0X
11:   ;save .detected in RA8
12:   C0 BRNZ RA1, .detected
13:   ;save .multiply in RA8
14:   C0 BR .multiply
15:   NOP 2
16:.multiply:
17:   ;repeat steps 8-13
```

(b)

Figure 6: Example illustration of SMART-SC: (a) Original Xentium® assembly instructions, and (b) modified Xentium® assembly instructions with SMART-SC

in the original set of instructions in lines 2 to *.add* label and in line 7 to *.multiply* label (Figure 6(a)). To detect transient errors during branch to *.add* label in SMART-SC, first the intended branch label (i.e. *.add*) address is stored in reserved register $RA8$ (lines 2-4, Figure 6(b)). Later, after each label (for example, *.add* label), further instructions are inserted to generate the label address from program counter (PC) and store in register $C0X$ (line 8). Then using the stored values in $RA8$ and $C0X$, a signature register value is newly generated using XOR operation between the two values and stored in $RA1$ (line 9-10). This signature value in $RA1$ is then compared with 0. When an error takes place in the program counter or in any of the storage registers (i.e. $RA8$ or $RA1$), the signature register will contain non-zero value and enable the detection of one or more errors. For example, if a transient error affects the *.multiply* label in line 14 and changes it to *.add* label,

added instructions in SMART-SC (in line 13 and lines 8-10) will detect an illegal branch as non-zero value will be generated in register $RA1$ (Figure 6(b)). Upon detection of illegal branch instructions, the detection flag is set and the roll-back routine ($sw\_roll - back$) is invoked in *.detected* label (further details in Section III-C).

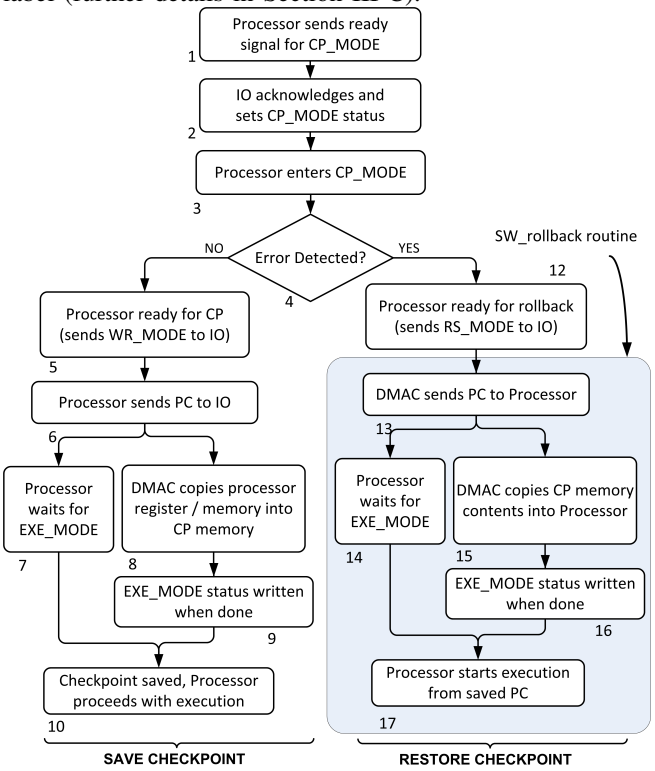

Figure 7: SMART-CP check-pointing mechanism

C. SMART-CP: Software Implemented Check-pointing

Mitigation of the detected errors (Sections III-A and III-B) in SMART-CP is performed using pre-compiled software implemented check-pointing and roll-back (i.e. saving the system state and restoring it when errors are detected) libraries incorporated through software modification. Since, the system architecture under consideration (Figure 1) does not provide with any timing or interrupt support to control executions contexts, the proposed technique employs a software handshake between DMAC and processor units to implement the check-pointing mechanism, as shown in Figure 7. The routine is initiated by the processor sending a status signal indicating that it is ready to perform check-pointing and waits for IO status CP_MODE to be written back by the DMAC unit (steps 1-2). When CP_MODE is seen by the DMAC unit, it notifies by writing CP_MODE back to the processor. When this status signal is acknowledged by the processor, it checks to see if one or more errors have been detected (through the $detected$ flag in the $modify.h$ library). When errors are detected, software implement roll-back routine ($sw\_roll - back$) is called up. Otherwise, the system proceeds with saving the current system state (steps 3-4).

When handshake sub-routine is completed, the Xentium® first writes the current program counter address on the IO port for DMAC to save it in a dedicated memory location within the separate memory unit during check-pointing (steps 5). Once it is acknowledged, the processor unit then waits for a routine completion status EXE_MODE. The waiting essentially manifests software based interrupt routine from processor side, which is required for the DMAC unit to allow saving or restoring the system state (steps 7, Figure 7). During this time, the DMAC unit saves the register values and memory contents from the processor to the CP memory.

When the system state is rolled back due to one or more transient errors detected ($sw\_roll - back$, Figure 7), the DMAC unit writes back to the processor the saved PC address and register/memory contents from the previously saved check-point (steps 12-15). Similar to check-point save routine, roll-back routine also implements software based handshake between the DMAC and processor, while the DMAC can communicate the system state back to the processor. During this time, the processor waits for handshake signal EXE_MODE from the DMAC. When the DMAC completes the transfer of system states, it writes the signal EXE_MODE back to the processor. Upon acknowledgement of this signal, the processor then proceeds with the updated system state.

Algorithm 1 : Check-point routine insertion in approximate intervals

```
1: Assume: Desired check-pointing interval, C cycles
2: for statement type != conditional branch statement do
3:     Insert dummy return routine
4:     Check approx. Xentium® cycle count
5:     if cycle count <> C then
6:         Insert branch instruction to CP routine
7:     else
8:         continue
9:     end if
10: end for
```

Since execution context in Xentium® cannot be controlled through hardware interrupts, the check-pointing routine (Figure 7) is then inserted in the original software description (at assembly-level) in suitable locations implementing a quasi-periodic check-pointing mechanism. Algorithm 1 shows how this is carried out for a given application. As can be seen, the check-pointing routine insertion involves finding convenient locations within the software description (line 2). In particular, conditional branch instructions (e.g. $BRNZ$, BRZ etc.) are skipped, since such statements can vary the program flow depending on the conditions. Dummy return statements are inserted in these potentially convenient locations to find out if the cycle counts approximately meet the given check-pointing interval C (lines 4-9). These steps are continued until the application is sufficiently check-pointed.

IV. Experimental Results

The error detection and tolerance efficiency of the proposed prototype tool is demonstrated. This is then followed by system overhead evaluation of the tool-generated software specification in terms of performance, energy

Table I: Benchmark applications with injected SEUs in different instructions

App.	Avg. Cycles/run	% SEUs in STW / LDW	% SEUs in Branch	% SEUs in Arith / Logical	% SEUs in Others	% Incorrect execution (without SMART)	% Incorrect execution (with SMART)
FIR	74953217	34.7	14.3	41.4	9.6	66	13
DWT	98353826	33.4	16.4	42.1	8.1	61	14
FFT-fixed	103505361	35.3	15.6	39.7	9.4	63	11
WHT	97526718	39.3	13.6	37.5	9.6	60	15
Viterbi	89962118	38.9	17.3	35.2	8.6	63	13
JPEG	311822971	41.3	15.9	34.5	8.3	59	12

and memory footprint. Finally, SMART is comparatively evaluated with other reported techniques.

A. *Effectiveness of Error Detection and Tolerance*

A number of experiments are carried out in Xentium® ISS environment using different signal/image/data conditioning and processing benchmark applications to evaluate the effectiveness of the proposed SMART tool with system architecture (Figure 1). To demonstrate the effectiveness of th tool-generated modified software description, initially full data duplication with U_{NT}=0 is assumed (Section III-A). For each application, an approximate total of 5000 transient errors are injected in consecutive runs using single-event upset (SEU) based model (Section II-B). An arbitrary error probability of 10^{-12} (SEU per cycle) is assumed. Similar error injection has also been carried out in [2], [15].

Table II: SEU locations and their impact in software

Location	Detection and Impact
Memory	SEUs in accessed memory locations are detected; SEUs in unaccessed memory locations will have no effect (i.e. masked) at application-level
Memory Address	SEUs in accessed memory addresses are detected due to difference in computations in SMART-ED; some SEUs can generate segmentation fault leading to incorrect execution
Datapath	If SEUs cause instructions or operands to be changed, they are detected. If, however, modified instructions have non-equivalent operands, it will cause abrupt termination and incorrect execution
Registers	All SEUs in registers affect currently operating data and are detected. However, SEUs can also be masked if value change does not affect arithmetic or logical operation
Control Flow	SEUs that cause improper branching within known known control flow will be detected. However, SEUs that change the jump label to unknown labels will be undetected, leading to incorrect executions

Table I shows the impact of SEU injections on these benchmark applications. The average execution cycles per application for a given test input sequence is shown in column 2, while the percentage of injected SEUs in different instructions are shown in columns 3-6 (determined after application profiling through output trace followed by error injection, as explained in Section II-B). Columns 7-8 compare the percentage of incorrect executions without and with SMART technique. From Table I two major observations can be made. Firstly, it can be seen that depending on the nature of applications the percentage SEUs injected in different instructions can vary. However, the injected SEUs show a general trend of higher SEUs being injected in load/store (i.e. LDW/STW) and arithmetic/logical instructions (which is expected as DSP applications are highly computationally intensive in nature). For example, for memory and computationally intensive JPEG application the highest number (41%) of SEUs are injected in the load/store memory instructions, while about 34% SEUs are injected in the arithmetic/logical instructions. The branch instructions are subjected to the next highest percentage of SEUs injected, which varies depending on the nature of application. With the given SEUs injected, the second observation is that proposed SMART technique can significantly reduce the number of incorrect executions (by up to 80% in the case of FIR application) through software implemented error detection and tolerance technique (Section III). This is because, proposed technique can effectively detect the SEUs injected and mitigate the impact of these SEUs through application check-pointing, enabled through software modification (in SMART-CP). However, note that up to 15% of the cases can still lead to incorrect executions (in the case of WHT application) using SMART technique. This underlines the limitations of the proposed software-only technique (Section III). Table II summarizes these limitations, showing the possible SEU locations and their impact in the form of masking (no effect of error), detection or even being undetected. As can be seen, the proposed software modification approach can deal with errors incurred in various parts of the processor (datapath, registers and memory). The errors in these parts can be effectively detected if illegal control or memory referencing is not caused. However, when such illegal control flow or memory referencing is caused, incorrect execution is generated by the software descriptions (as shown in col 8, Table I).

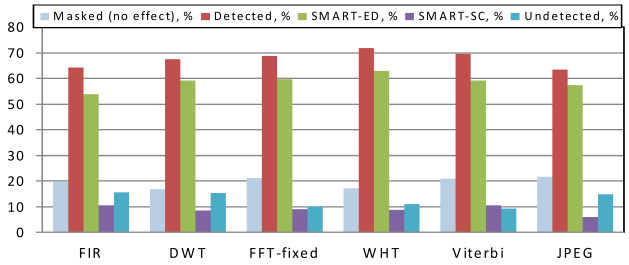

Figure 8: SMART detection efficiency

To demonstrate the effectiveness of error detection through SMART technique, Figure 8 shows the effective percentage of masked (i.e. errors with no effect), detected (both by SMART-ED and SMART-SC) and undetected SEUs through SMART technique in the benchmark applications (Table I) with $U_{NT} = 0$ (i.e. full duplication of all data). From Figure 8, two observations can be made. First observation is that maximum number of injected SEUs (up to 74% for WHT application) are detected through the proposed technique, while 17% SEUs can get masked, summing to an effective total of 91% errors being either detected or masked. Out of the detected SEUs, up to

85% are detected through the SMART-ED (Section III-A), while the rest of the SEUs are detected by SMART-SC (Section III-B). This is expected as SEUs injected in load/store and arithmetic/logical instruction dominate (Table I). Note that up to 17% of the SEUs cannot be detected depending on where SEUs are injected due to reasons explained in Table II.

B. Performance and Memory Footprint

The inclusion of data duplication, dynamic signature generation and check-pointing through software modification (Section III) imply performance overheads. This is because SMART-ED generates controlled repetitions of computations through normalized usage threshold, U_{NT} (Figure 4(b)), while SMART-SC requires signature generation and comparison cycles as well (Figure 6(b)). Furthermore, SMART-CP requires processor to software implemented check-pointing, which incurs more overheads.

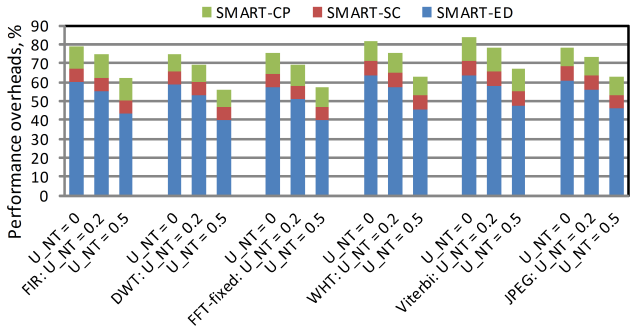

Figure 9: SMART performance overheads

To explore the performance overheads incurred due by SMART generated software modification, Figure 9 shows the normalized performance overheads of different benchmark applications with varied duplication control through normalized usage threshold (U_{NT}). An approximate check-pointing interval of $C \approx 4 \times 10^6$ cycles is chosen arbitrarily for the applications in comprehensive error injection environment (Section II-B). As can be seen SMART technique can result in a reasonable performance overhead for incorporating SEU detection and tolerance depending on the application and its nature of computation. For example, up to 83% performance overhead is incurred by Viterbi application (when all variables and data are duplicated, i.e. $U_{NT} = 0$). Note that although the proposed technique employs duplicate storage and computations, the performance overhead is contained. This is because the post modification compilation steps can employ instruction-level parallelism (a common feature in high-performance DSP processors) to reduce the performance overheads. Comparing the contributions of different SMART steps (Figure 3), it can be seen that the highest performance overhead (about 65% for the WHT decoding application) is caused by SMART-ED (Section III-A). SMART-CP is the second largest contributor for the performance overheads (with up to 13% overhead for the JPEG application). This can, however, vary depending on the approximate check-pointing interval assumed (the variation of CP sizes and its impact is studied in detail in [18]). SMART-SC contributes the lowest performance overheads for the given check-pointing interval when compared with the other steps. As expected, with higher normalized usage threshold, the duplication in the original application (through SMART-ED) can be controlled and the performance overheads show a trade-off. For example, when the duplication is only carried out for variables/constants that have more than 50% normalized usage (i.e. $U_N \geq (U_{NT} = 0.5)$), the performance overhead reduced to as low as 56% for DWT application. Similar trend can also be observed from other applications. However, reduction of duplication comes with a decrease in the detection coverage as well. For example, in the case of FIR application, the detection coverage (detected and masked) can reduce to 64% when $U_N \geq (U_{NT} = 0.5)$ (compared to 91% when $U_{NT} = 0$).

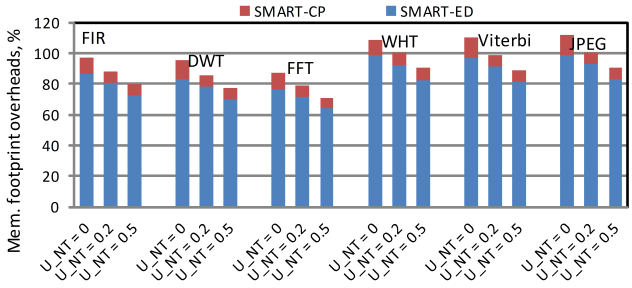

Figure 10: SMART memory overheads

Due to software modification, software descriptions can also expand with increased memory footprint. Figure 10 shows the comparative memory footprints of different applications, showing contributions from different SMART steps. Since SMART-SC does not use memory resources, it is not shown. As expected, SMART-ED gives the highest memory overheads due to duplicated data components and related comparison instructions for detection of SEUs (Section III-A). Comparing the different applications, it can be seen that a reasonable 113% memory footprint overhead is caused by SMART technique in Viterbi and WHT applications, for example. This is because these applications use comparatively larger amount of data storage components (variables and data value holders) at high-level description, which is further duplicated by the proposed technique (SMART-ED, Section III-A). From Figure 10, it can also be seen that with controlled duplication through SMART-ED using normalized usage threshold (Section III-A), memory footprint can be reduced.

C. Comparative Evaluation

To further demonstrate the effectiveness of the proposed SMART technique implemented in a prototype tool, it is comparatively evaluated against the following reported techniques that can be implemented for error detection and correction for COTS components: duplex processing and comparison [3], triple modular redundancy [8] and high-level software transformation technique [15]. Hardware- or compiler-level designs are not feasible for the COTS component under consideration and hence are not considered in the comparisons. Table III shows the comparisons with

various attributes (column 1) and their respective properties of each technique, including SMART (columns 2-5).

Table III: Comparative evaluations of SMART

Attribute	Duplication [3]	TMR [8]	SW Transform. [15]	**SMART** (proposed)
Detect. coverage	high ($\approx 100\%$)	high ($\approx 100\%$)	low ($\approx 51\%$)	high ($\approx 74\%$)
Error tolerance	none	high (time-limited)	none	high
Area overhead	$> 100\%$	$> 200\%$	none	DMAC+CP memory
Perform. overhead	$\approx 12\%$	$\approx 15\%$	$\approx 70\%$ (with ILP)	$\approx 83\%$ (with ILP)
HW design	compar. and sync.	compar. and sync.	none	none

As can be seen, when error detection is considered both duplication and TMR techniques have high detection coverage ($\approx 100\%$) as any error that leads to incorrect execution is detected at the output comparator (row 2). The software transformation technique [15] suffers from poor detection coverage of only 51%, since the transformation only considers computation related errors without taking into account the larger proportion of errors incurred in memory. The proposed technique gives higher detection coverage as errors in processor datapath, registers and memory are taken into account during software modification (through SMART-ED and SMART-SC, Section III). Note that both duplication [3] and also software transformation [15] techniques cannot tolerate the detected errors as no tolerance mechanism is incorporated upon detection (row 3). Our proposed technique gives high error tolerance through software implemented check-pointing approach (giving up to 80% less incorrect execution than the original software description without any modification). The TMR approach [8], however, achieves a higher error tolerance (which is mission time-limited) at the cost of high area overhead (row 3). The proposed SMART technique, however, incurs performance overhead, similar to [15], of up to 83% since extra computation and storage instructions are incorporated for error detection and tolerance (see Section IV-B). Finally, when design effort is considered, the proposed SMART technique outperforms duplication or TMR approaches, as no extra hardware customization or design efforts are needed. This underlines a significant advantage of the proposed technique, making it particularly suitable for COTS components with limited or no hadrware- or compiler-level access.

V. Conclusions

A software only and architecture-independent transient error detection tolerance technique implemented in a prototype tool (called SMART) was presented. The tool incorporates controlled data duplication and dynamic signature comparison in the modified software description to detect transient errors. To achieve effective error tolerance software implemented check-pointing is carried out through further software modification. We showed that SMART can achieve high detection coverage and significantly reduce the incorrect executions (by up to 80%), while providing a trade-off between design efforts and performance overheads with controlled duplication. Due to software-only nature, the proposed technique is particularly suitable for systems designers with limited access to hardware or compiler modifications.

References

[1] P. Bernardi *et al.* A new hybrid fault detection technique for systems-on-a-chip. *IEEE Trans. Computers*, 55(2):185–198, Feb. 2006.

[2] S. Campagna and M. Violante. An hybrid architecture to detect transient faults in microprocessors: An experimental validation. In *Proc. Design, Automation & Test in Europe (DATE)*, pp.1433–1438, 2012.

[3] P. T. De Sousa and F. P. Mathur. Sift-out modular redundancy. *IEEE Trans. Computers*, 27(7):624–627, July 1978.

[4] A. Ejlali *et al.* Combined time and information redundancy for seu-tolerance in energy efficient real-time systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(4):323–335, April, 2006 2006.

[5] R. han Kim *et al.* 22 nm technology node active layer patterning for planar transistor devices. In L. H. J. and M. V. Dusa, editors, *Proc. Optical Microlithography XXII.*, vol. 7274, pp.72742X–72742X–6, 2009.

[6] J. Hu *et al.* Compiler-directed instruction duplication for soft error detection. In *Proc. DATE*, pp.1056–1057, Mar. 2005.

[7] P. Kudva and J. Rivers. Balancing new reliability challenges and system performance at the architecture level. *IEEE Design and Test of Computers*, PP:99, 2012.

[8] L. Longden *et al.* Designing a single board computers for space using the most advanced processor and mitigation technologies. *ESA Publications*, 507:313–316, 2002.

[9] S. Mitra. Globally optimized robust systems to overcome scaled CMOS reliability challenges. In *Proc. DATE*, pp.941–946, 2008.

[10] N. Oh *et al.* ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Computers*, 51(2):180–199, Feb. 2002.

[11] N. Oh *et al.* Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliability*, 51(1):63 –75, mar 2002.

[12] M. Pignol. DMT and DT2: Two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers. In *Intl. Online Test Symposium*, pp.203–212, 2006.

[13] F. Rashid *et al.* Fault tolerance through re-execution in multiscalar architecture. In *Proc. Dependable Systems and Networks (DSN)*, pp.482–491, 2000.

[14] R.A. Shafik *et al.* System-level design optimization of reliable and low power multiprocessor system-on-chip. *Microelectronics Reliability*, 52(8):1735–1748, 2010.

[15] M. Rebaudengo *et al.* System safety through automatic high-level code transformations: an experimental evaluation. In *Proc. DATE*, pp.297 –301, 2001.

[16] G. Reis *et al.* SWIFT: software implemented fault tolerance. In *Intl. Symp. Code Generation and Optimization (CGO)*, pp.243–254, Mar 2005.

[17] S. R. Shenoy and A. Daniel. Intel architecture and silicon cadence: the catalyst for industry innovation. Tech. report, Intel Corp., April 2009.

[18] I. A. Troxel *et al.* Reliable management services for COTS-based space systems and applications. In *Proc. Intl. Conf. on Embedded Systems & Applications*, pp.169–175, 2006.

[19] Xentium: High performance DSP processor. Last accessed 14/03/2013. http://www.recoresystems.com/products/dsp-accelerator-ip/xentium-ip-core/.