SEU Mitigation Techniques for Microprocessor Control Logic*

Ganesh T S†, Viswanathan Subramanian and Arun Somani
Dependable Computing and Networking Laboratory
Iowa State University
{*ganeshts*}@ambarella.com, {*visu, arun*}@iastate.edu

Abstract

The importance of fault tolerance at the processor architecture level has been made increasingly important due to rapid advancements in the design and usage of high performance devices and embedded processors. System level solutions to the challenge of fault tolerance flag errors and utilize penalty cycles to recover through the re-execution of instructions. This motivates the need for a hybrid technique providing fault detection as well as fault masking, with minimal penalty cycles for recovery from detected errors. We propose three architectural schemes to protect the control logic of microprocessors against Single Event Upsets (SEUs). High fault coverage with relatively low hardware overhead is obtained by using both fault detection with recovery and fault masking. Control signals are classified as either static or dynamic, and static signals are further classified as opcode dependent and instruction dependent. The strategy for protecting static instruction dependent control signals utilizes a distributed cache of the history of the control bits along with the Triple Modular Redundancy (TMR) concept, while the opcode dependent control signals are protected by a distributed cache which can be used to flag errors. Dynamic signals are protected by selective duplication of datapath components. The techniques are implemented on the OpenRISC 1200 processor. Our simulation results show that fault detection with single cycle recovery is provided for 92% of all instruction executions. FPGA synthesis is performed to analyze the associated cycle time and area overheads.

1. Introduction

Developments in VLSI technology and numerous architectural innovations have fueled advancements in microprocessor design. Increasingly miniaturized systems and higher frequencies of operations, as well as utilization in hazardous environments, make microprocessors susceptible to faults. One of the most important factors threatening the reliability of future computer systems will be soft errors [15]. An error can simply be defined as an unwanted change in the logic value of a signal. Soft errors, also referred to as transient errors or single event upsets (SEUs), are caused by transient pulses resulting from radiation striking the surface of the silicon in the processor. As scaling in VLSI technology continues into the nanometer regime, both memory elements and combinational gates become susceptible to soft errors due to the transient pulses induced by radiation having durations often higher than the gate propagation delays. This electric pulse can propagate without masking and may result in an error (soft error) at the application level, hence affecting the system behavior. Further, higher clock speeds decrease the cycle time, increasing the probability that a soft error is latched. These trends imply that future digital systems need to be protected against both single event transients (SETs) and SEUs [5][12].

Various techniques at many levels of the design hierarchy have been developed to protect microprocessors against failure due to faults. Fault tolerance can be achieved by usage of redundancy in the spatial, temporal or information domain. Spatial redundancy is achieved by carrying out the same computation on multiple independent functional units at the same time. Errors are brought out on comparison of the redundant results. A majority selection scheme can be used to obtain a correct answer for systems implementing triple modular redundancy (TMR) or higher redundancy under certain conditions of failure. In case there are only two redundant units available, the computation must be restarted if the two results do not match. Spatial redundancy techniques are increasingly becoming popular in the fault tolerance community due to higher amounts of silicon real estate available to designers. Redundancy in the time domain works by repeating the computation on the same hardware multiple times. Information redundancy involves the storage of extra bits (information) to cross check whether the

*The research reported in this paper is partially supported by NSF grant number 0311061 and the Jerry R. Junkins Endowment at Iowa State University.

†Currently with Ambarella Corporation, CA

data retrieval has been free of errors. All such techniques can be said to offer fault tolerance by either performing fault detection coupled with recovery or providing fault masking.

There are numerous SEU tolerant architectures which have been proposed from the research community. AR-SMT [13] proposes using the multi threading capability of modern processors to execute the program and a duplicate of the program in parallel as two threads. DIVA [1] uses spatial redundancy by providing a separate, slower pipeline processor along side the fast processor. The Selective Series Duplex architecture [7] consists of an integrity checking architecture for superscalar processors that can achieve fault tolerance capability of a duplex system at much less cost than the traditional duplication approach. Soft error detection and recovery in the IBM Z990 is discussed in [9]. Many such proposed techniques protect the core datapath against faults. However, the aspect of the control logic is largely ignored. This is due to the fact that control signals are not easily amenable to protection using the usual fault tolerance techniques. Schemes which provide for fault detection and correction higher up in the design hierarchy usually have higher penalties for recovery in comparison to fault masking and schemes implemented at a lower level. Protection of FSM based control logic by error masking has been discussed in [3]. A signature caching scheme was proposed in [6] to detect SEUs in the control logic of complex microprocessors. The ReStore architecture [16] uses checkpointing and rollback to recover from soft errors. The rollback is done based on certain abnormal events such as exceptions and incorrect control flow. This paper presents a methodology to obtain high coverage with relatively low hardware overhead and performance impact with respect to control signal faults. In addition, the presented methodology can also be used in conjunction with other schemes protecting the system against datapath faults, since the control and datapath are considered as separately.

We classify control signals as static and dynamic depending on variations in their values with the execution of the instruction at different points of time. Static control signals are further classified as instruction dependent and opcode dependent. We propose a scheme of three integrated techniques to protect different types of control signals, and to overcome the shortcomings of the present control logic protection schemes. It operates by providing for both masking as well as detection and correction of faults.

The remainder of the paper is organized as follows. Section 2 gives an overview of the proposed technique. Sections 3, 4 and 5 present the details of the the schemes to protect three different types of control signals. Section 6 presents the implementation details and performance analysis.

2. SEU Mitigation Techniques for Microprocessor Control Logic: An Overview

We tackle the task of fault tolerance by classifying control signals at the design stage on the basis of their attributes and providing schemes for their protection. This section deals with microprocessor control logic and their classification for the purpose of the proposed technique followed by a brief sketch of the component schemes.

2.1. Microprocessor Control Logic

All modern day microprocessors are based on a pipeline structure comprised of fetch, decode, execute, memory access and write-back stages. The control signals of a fetched instruction are generated in the decode stage and traverse through the pipeline along with the instruction. The purpose of the proposed technique is to tackle SEUs / SETs in the control logic segment of the datapath, as shown in Figure 1, without making any assumptions about the number of pipeline stages.

The control signals of an instruction can be broadly classified as being either static or dynamic. Static control signals are those which remain invariant irrespective of the state of the processor when the instruction executes. For example, the register write signal for an instruction is not dependent upon the processor state, but only on the instruction. Hence, it is a static control signal. Dynamic control signals, on the other hand, are dependent upon the state of the processor when the instruction executes. For example, the signal indicating a taken branch could be dependent on whether the present contents of two registers are equal.

Static control signals are further classified as being purely dependent on the instruction itself or only on the instruction opcode. The select control signal of the multiplexer at the input of the ALU is an example of the former. The select line could choose the contents from the register file, or from the immediate field or from the forwarding logic. The control signal for the register file write is dependent upon the opcode only. Hence, it is an example of a static control signal dependent upon instruction type.

It would be difficult to design a single scheme to protect all the above types of control signals. Hence, a technique consisting of three integrated schemes is proposed to provide fault tolerance for the control logic, one for each type of control signal. The next subsection provides a brief sketch of the components of the proposed technique.

2.2. Component Schemes of the SEU Mitigation Technique

A common quantitative principle used in computer architecture is the concept of locality of reference. It refers to

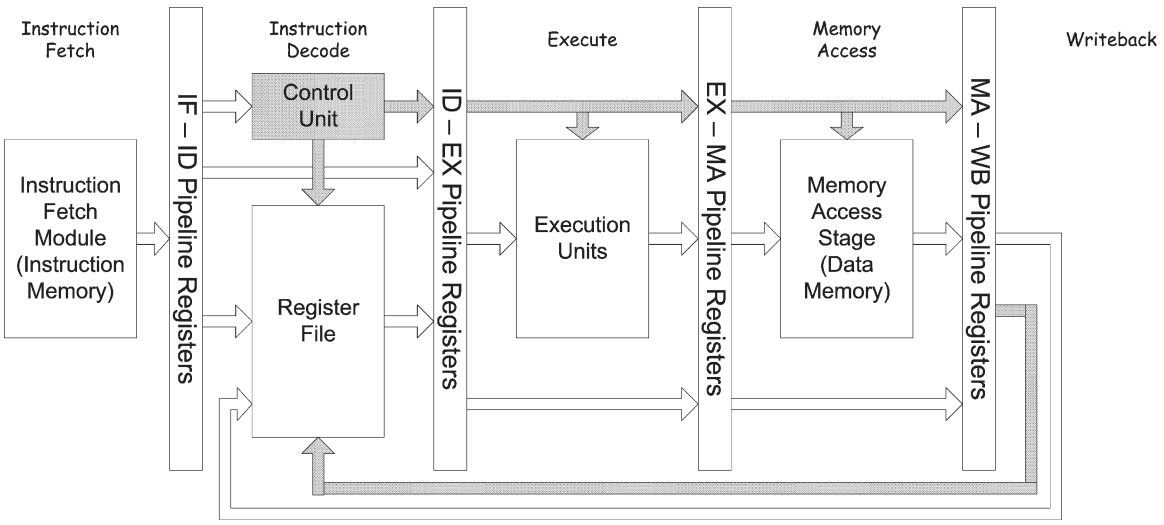

Figure 1. Typical 5-stage microprocessor pipeline with the addressed vulnerabilities shown thatched

the observation that 90% of the program execution time is spent in 10% of the code [10]. This is due to the fact that most common workloads are heavily oriented towards looping structures. An implication is that there would be a high probability that an instruction at a particular address would be processed multiple times in the course of a program's execution. The scheme for protecting instruction dependent static control signals works on the assumption that an SEU / SET will not affect a particular control signal (say, the write enable signal for the register file) in two out of three consecutive iterations of the same instruction. If each control bit of a particular instruction from the previous two iterations is stored, we would be able to maintain a history of these signals. In other words, the control signal bit is cached in the pipeline stage in which it is utilized. At any given point of time from the third iteration of an instruction's execution, we would have three values of the signal to choose from. A majority function can be implemented on the three values to arrive at a TMR based decision. The technique can provide complete fault masking from the third iteration of a loop, and detection from the second iteration.

Static opcode dependent signals are easier to protect since a static cache indexed by the opcode can store the different control signals in each pipeline stage. A comparison of the indexed entry with the present value of the control signal can help in detecting faults.

Dynamic signals are the most difficult to protect of the three different control signal types. Duplication of the component producing the dynamic control signal and comparing their results would detect faults.

A high level view of the working of the component schemes is shown in Figure 2. It traces the execution of three sample instructions in a particular program. The instruction at PC value 100H, (add $r3,$r0, $r0), is executed only once, while the instruction at PC value 160H, (addi $r4, $r4, -1), is executed once every time the program goes past the 'LOOP' label. The third instruction under consideration (bf LOOP) is at PC value 180H, and modifies the PC value to enable program execution from the 'LOOP' label if the flag bit in the status register is set by the instruction at 176H.

As the program executes for the first time, the instruction at PC value 100H is processed. The static opcode dependent control signals are protected by fault detection and correction since the datapath already has a distributed cache of control signal values corresponding to different opcodes. To illustrate the working of this component scheme, let us consider the register write signal in the writeback stage. In this particular case, the opcode is that of an R-type instruction. As soon as the instruction reaches the writeback stage, the opcode value indexes into the writeback stage control cache and fetches all the relevant control signals. The fetched control signals are compared against the values propagating along with the instruction through the pipeline. Any mismatch triggers a recovery mechanism, details of which are presented in a later section. It must be noted that control signals which vary from instruction to instruction can't be protected by this component scheme.

As the program continues executing, the instruction at PC value 160H is processed for the first time. The static instruction dependent control signals protection scheme begins functioning, and recognizes the fact that the instruction is being executed for the first time. Let us consider the destination register number in the register writeback stage. It is

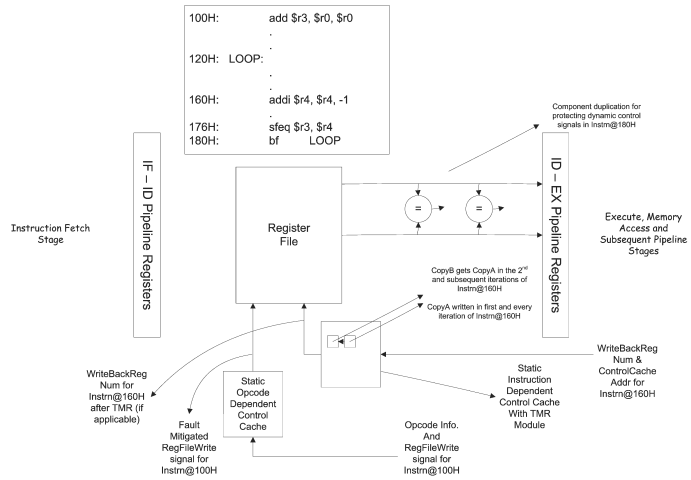

Figure 2. High level view of the component schemes of the SEU mitigation technique

utilized for the writeback, and also written into the instruction dependent control cache at the address corresponding to the instruction's PC value.

The program continues executing and reaches the instruction at PC value 180H. This instruction reads the value of the status register, whose fields are dynamic control signals. The values are generated by the instruction at PC value 176H. Either a second copy of the status register is utilized to decide the outcome of this instruction, or the component utilized by the instruction at 176H is duplicated. Thus, dynamic control signals are handled by selective duplication of processor components.

For the purpose of this illustration, let us assume that the branch is taken and control shifts back to the instruction at the 'LOOP' label. The instruction at 160H is executed again, and this time the logic associated with the instruction dependent control caching scheme recognizes that the instruction at that particular PC value has been executed before. The control signals are compared with the signals from the previous iteration and a mismatch triggers a recovery mechanism, just as in the static opcode dependent control signals protection scheme. This is done only for the second execution instance, in order to improve fault coverage. In addition to this comparison, the control signals of the instruction during this iteration are written into the relevant cache. The previous contents, instead of being overwritten, are shifted along, in such a way that the third execution instance of the particular instruction has at its disposal the history of control signals from the two previous iterations. A TMR based decision is made possible from the third iteration onwards with the aid of a moving window of the history of the control signals.

It must be noted that the additional hardware involved, such as the gates involved in the TMR based decision, are also protected against faults due to the inherent fault masking nature of the component schemes, since they result in erroneous masking only when two components become faulty at the same time. Further, the additional cache memory instantiated is assumed to be protected by error correcting codes (ECC). ECC is a common feature in all modern RAM libraries, and is hence not elaborated further in this paper.

3. Opcode Dependent Control Caching

The majority of the control signals in a microprocessor datapath fall under the static instruction dependent category. Opcode dependent control signals form the next biggest set. As outlined in a brief sketch in the previous section, the protection for the opcode dependent control signals is facilitated by the addition of a distributed storage of control signals indexed by the opcode. The term 'distributed' here implies that each pipeline stage has a cache which stores only the control signals to be processed in that particular stage. Thus, the size of the cache in each stage will be dependent on the number of control signals in that stage.

The opcode of the fetched instruction is processed to yield the address in the opcode control cache. This address flows through the pipeline along with the instruction. The control signal and the contents read from the cache are compared, and any mismatch implies a fault in the control signal. Before the faulty control signal can be used, the pipeline should be frozen. This freeze lasts long enough for the transient or upset to die down. This implies a strategic placement of the cache contents for each signal such that the fault detection happens in the stage prior to the usage of that particular control signal. In order to ensure that the opcode itself is free from transient errors, it is treated as an instruction dependent control signal and is protected by the

static instruction dependent control signal scheme outlined in the next section.

It must be noted that this scheme is not dependent upon the programs operating in loops. It offers protection to all the executed instructions. On the downside, there is no fault masking and the scheme can't prevent penalty cycles while recovering from faults.

4. Instruction Dependent Control Caching

A brief sketch of the static instruction dependent control signals protection scheme has been already outlined. In this section, we discuss the finer details of the scheme. These include the logic associated with determining whether an instruction under consideration is part of a loop, and if so, determining the number of times it has been executed before. It is obvious that the history of control signals can't be stored for the entire program, but only for a limited number of instructions. In other words, the cache organization for the control cache has to be decided. Another interesting aspect is the design of the distributed cache structure. These points are covered in the following subsections.

4.1 History Determination

Every cache entry in the static instruction dependent control caching scheme has a finite state machine (Figure 3) associated with it to maintain the history details.

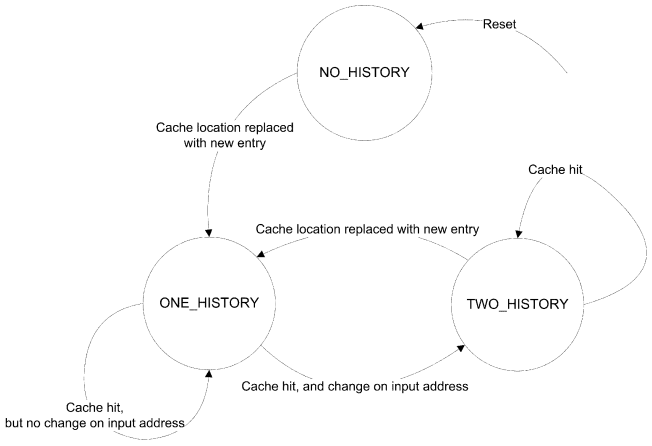

Figure 3. FSM for tracking the cache entry history

On the receipt of a reset signal, all the cache entries set themselves in the NO_HISTORY state. The FSM lies dormant till its corresponding address is activated by a write in the cache entry. When a new address is input, the fact that there is no history available is sent out in the same cycle. However, as the entry registers itself, the fact that its first iteration is underway is indicated by the update of the entry at the writable address to ONE_HISTORY. It is very much possible that the fetch address remains in the same state for multiple cycles, due to stalls or other reasons. For this reason, a register holds the address output in the previous cycle. A hit which was not present in an earlier cycle cause an update in the state provided the output address is not the same as the previously stored address. The saturation of the state machine in any of the states is prevented by the fact that the cache entry can always be replaced by a newly input address.

4.2 Cache Organization Determination

The control cache address and history generation module implements the cache organization. It is placed in the instruction fetch stage of the pipeline since the instruction fetch address is available in this stage. The same address can be used to index into the created module to output the control cache address and the history status for the corresponding instruction.

There are numerous options available for the cache structure. The tag and address tracking mechanism could be either a direct mapped or a fully associative structure. Commonly used replacement policies in the fully associative structure are First In First Out (FIFO) and Least Recently Used (LRU). The direct mapped structure is chosen for its implementation simplicity, while the fully associative structure is chosen due to the fact that all the entries provided can be utilized.

The determination of the optimum caching type and size is done through profiling of the workloads. The different cache types were modeled in Simplescalar [2] and SPEC 2000 benchmarks were processed to determine the amount of coverage provided by the cache. Simulation results and analysis are presented in Section 6. The results indicate that the performance of the FIFO based fully associative structure and the direct mapped structure are very close to each other. The LRU policy in a fully associative scheme is unsuitable for the caching requirements. While coverage for the FIFO structure for fault detection alone is marginally better than the direct mapped structure, the hardware requirements and performance penalty for the FIFO structure are much more. The size of the cache is directly related to the amount of history to be kept track of, which in turn directly affects the provided coverage.

4.3 Distributed Cache Structure

The control cache entry address and history generated above are passed down the pipeline and utilized in each stage. Each control signal bit to be protected has a structure similar to the one outlined in Figure 4 placed just before the

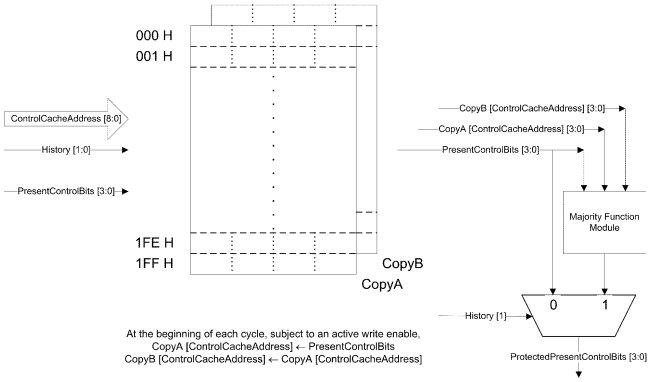

Figure 4. A 512 entry control caching structure for a 4-bit control signal

point of utilization in its own pipeline stage. The structure conceptually consists of two planes of memory bits organized in a fashion to facilitate flow of a particular bit from the top to the bottom plane. This flow occurs whenever the control signal for the instruction at that particular cache address location is written. The protection is offered by reading the two sets of bits at the particular address, combining it with the input bits and using a majority function gate to generate a new set of control bits. Depending on the history for that particular instruction, either the original control signal or the newly generated control bits are utilized in the datapath. This provides the fault masking capabilities of the scheme. The fault detection and recovery capabilities of this scheme are the same as the one to be outlined in Section 3.

5. Dynamic Control Signal Protection Scheme

Dynamic control signals are the most difficult to protect due to their inherent unpredictability. Fortunately, they comprise the least number of control bits. The scheme to protect these bits relies on selective duplication of the source. Analysis of an industry standard microprocessor for embedded applications [8] reveals that only 17% of the control bits are dynamic in nature. These signals do not originate from the control unit, but are produced as a result of some operation in the datapath (such as branch or exception related signals, or signals relating to stalls) or some external source (such as trap and system call signals).

For example, the signal indicating a taken branch might be dependent on the value in the status register. Duplicating this status register (or the component writing to the status register) yields two values, which can be compared to determine faults (before writing to the status register itself), if any. Stall detection logic can again be managed with minimal redundancy in the form of duplication of comparators and source signals. External signals can be made to propagate on two separate lines and their integrity can be ensured before utilization in the datapath.

At the expense of some penalty cycles, the execution can be frozen till the concerned signals match each other. This scheme offers protection to all the executed instructions, irrespective of the looping nature of the program. There is no fault masking, and detected faults invariably lead to penalty cycles. However, it must be kept in mind that in comparison with existing schemes, the number of penalty cycles for protecting the same amount of logic is reduced by a considerable amount. The exact reduction in the number of penalty cycles can be determined by evaluating the ratio of the number of control signal bits protected by complete fault masking to the total number of control bits.

6. Implementation and Analysis

The proposed technique consisting of the three integrated schemes has been analyzed after implementation using a two-phase approach. The first phase involved the identification of the type and size of the control cache for protecting the static instruction dependent control signals. The second phase involved the implementation of the modified architecture using an industry-standard RTL processor model. Fault coverage analysis is performed by fault injection studies on the RTL model. FPGA Synthesis of the RTL models with different fault tolerance features added is also performed to evaluate the overheads associated with the scheme.

6.1 Workload Profiling for Static Instruction Dependent Control Cache Design

The control cache for static instruction dependent control signals can be modeled in the Simplescalar tool set. The designed cache structure is either fully associative or direct mapped in nature. For a fully associative structure, the entire program counter value must be used to tag the cache. This enables easy identification of the cache entry to use. Only the address of the entry needs to be passed on to the succeeding pipeline stages, and not the program counter value itself. The tag bits need not be duplicated in the entries in the other pipeline stages. In a direct mapped structure, only the bits of the PC which are not used to index into the cache need to be stored.

These cache-like structures are modeled wherein the option of the number of entries in the cache as well as the replacement policy to use (LRU or FIFO) can be set by the user. The proposed technique offers protection by masking only when a particular instruction is being executed for the third time or higher, and the entries are still in the cache.

However, it can detect and correct a fault in the second iteration itself. The developed model keeps track of the number of such instructions (the 'hits' in the cache) and also the total number of requests. It is a simple task, then, to determine the percentage of instructions for which the technique could actually afford protection, and the percentage of instructions which went unprotected, for different types and sizes of the cache.

In order to evaluate the control cache size requirements, precompiled Alpha binaries for six different integer benchmark programs were chosen from the SPEC2000 suite. The Simplescalar tool set was set to execute in the Alpha ISA mode and 1 billion instructions in each of the benchmarks were processed, after skipping the startup section, to yield the protection rate using the proposed scheme. It was determined that the FIFO policy is the best replacement policy. The direct mapped structure performs consistenly well in all the benchmarks, and provides close competition to the FIFO structure [4]. Depending on the amount of protection required, the number of cache entries can be decided. 512 entries for the control cache appeared to offer a good protection for all the benchmarks evaluated.

A caveat to the above analysis is that the technique presented is heavily dependent on the instruction set architecture (ISA) of the machine as well as the type of workloads. Some ISAs utilize a large number of instructions to perform basic tasks, which could in turn lead to a large number of instructions in each basic block. These types of programs will require a large number of entries in the control cache in order to ensure coverage or protection for the executed instructions. Another point to note is that the final coverage obtained would vary from workload to workload depending on the program and control flow structure. Programs which loop a large number of times are evidently offered more protection by the technique in contrast to programs with huge linear flow of control. The next subsection deals with the determination of the various fault tolerance metrics for the proposed scheme.

6.2 Fault Tolerance Metrics

The fault injection model is dependent on the fault model for which the fault tolerant hardware is designed. The control caching technique assumes a SEU model for the SER [14]. In this case, at any particular instance of time, only one particle hit can happen, and thus, only one particular signal can be affected. However, the duration of the upset can extend over multiple clock cycles. The frequency with which these upsets occur is dependent on the environment in which the processor is placed. The duration is dependent on the clock frequency of the design under consideration. A characteristic of SEUs is that they are random events and thus may occur at unpredictable times. In this work, we focus on the fault model called upset or transient bit-flip. These are the easiest to check for effects in fault injection simulations since they closely matches real faulty behavior [11].

There are two fault tolerant architectures designed for evaluation, one with the CAM FIFO structure for the static instruction dependent control cache, and the other with the direct mapped structure. Due to the simulation overhead involved [4], it would be better to go ahead with a pessimistic estimate of the fault coverage by consideration of the history state signal.

The history state output of the FIFO based CAM and the direct mapped module is related to the address currently input. The address trace of some of the SPEC 2000 benchmarks was input to a RTL model of the cache structure and the history state was recorded in each execution cycle. A history state of $TWO\_HISTORY$ was termed to afford complete masking, while a history state of $ONE\_HISTORY$ was termed to offer fault detection but no masking. The simulation of the history generation RTL model was carried out for both 512 and 1024 entries, and the results are presented in Figure 5. It can be seen that the direct mapped structure gives higher values for complete fault masking in comparison to the CAM-based FIFO structure for all benchmarks. An increase in the number of entries results in more coverage as expected, since the history of more number of instructions are kept track of. If detection and correction is considered, the trend is similar, except that the FIFO structure has a slight edge over the direct mapped structure. Some benchmarks have a significant number of unprotected instructions due to compulsory and capacity misses. The former is caused due to huge linear segments of control flow, while the latter is a result of the loop sizes exceeding the number of instructions which are being tracked. Quantitatively, it can be inferred from the graph that, on an average, a 512 entry direct mapped structure offers protection by complete masking and by fault detection and correction to 64% and 74% of all instruction executions respectively. The corresponding figures for a 1024 entry direct mapped structure are 69% and 82%. For a CAM-based FIFO structure, the values for a 512 entry cache are 52% and 79%, and for a 1024 entry cache, they are 59% and 91% respectively. Taking into consideration both the masking requirement as well as the fault detection requirements, a 1024 entry direct mapped structure appears to be a good trade-off, particularly when considering the hardware overhead of the CAM-based FIFO structure, discussed in a later subsection.

It must be noted that the above values refer to the lowest possible bound for the fault coverage. In practice, the generally high locality in program execution and natural masking of faults would ensure that the fault coverage would be much higher.

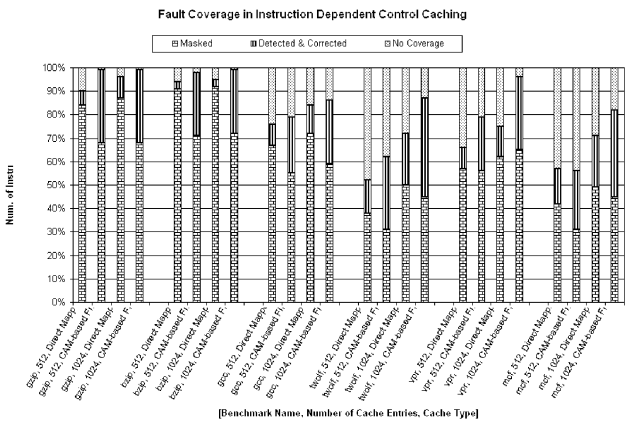

Figure 5. Fault coverage in different instruction dependent control cache configurations for various SPEC2000 benchmarks

For the opcode dependent control caching and dynamic control signal protection schemes, there is no fault masking, but fault detection and correction for all instruction executions. Due to the similar nature of control signals in the Alpha as well as the OpenRISC processor (on which the RTL implementation is performed 6.3), it is fair to assume that the relative number of different types of control signals remains approximately the same, more so, since both are RISC in nature. Under the above assumption, complete fault masking is available for 83% of all control signals (if opcode dependent control signals are also cached along with the instruction dependent control signals in the instruction dependent control caching scheme) for 69% of instruction executions on an average. Fault detection with correction covering all the control signals can be determined from the fact that the opcode dependent control caching and dynamic control protection is available for 100% of all instruction executions for 53% of all control signals, while the remaining 47% of the control signals are protected for 82% of all instruction executions. Denoting the percentage of opcode dependent and dynamic control signals by N_O and N_D, and the percentage of instruction dependent control signals by N_I, the percentage of instructions for which the opcode dependent and dynamic control signals are protected by NI_{ODD} and the percentage of instruction for which the instruction dependent control signals are protected by NI_{ID}, the nett fault coverage value F_C can be computed as shown in Equation 1.

$$F_C = \frac{((N_O + N_D)NI_{ODD} + N_I NI_{ID})}{100} = \frac{53X100 + 47X82}{100} = 92.07\% \quad (1)$$

Most fault tolerant architectures tend to flag an alert signal when there is no cause for alarm. This is termed as a 'false positive'. In the instruction dependent control caching scheme, there is no scope for a false positive. However, there is a possibility that the same control signal of the same instruction can be affected in two consecutive iterations of the loop. In this case, the first iteration would be protected by the earlier stored signals. However, the next set of iterations would suffer till the erroneous signal gets overwritten in the history. However, keeping in mind the probability of SEUs, this situation is very unlikely. Another related metric of interest is fault sensitivity. The proposed technique does not interfere with the fault sensitivity of any of the nodes. However, if it is determined that some of the nodes have a higher fault sensitivity, then, the technique can be applied to those nodes with a higher priority.

6.3 Control Caching Implementation in the OpenRISC 1200

The control caching technique is implemented on the RTL model of a microprocessor in order to evaluate both effectiveness and performance. We consider a specific implementation of the OpenRISC 1000 family of processors, namely, the OpenRISC 1200 processor [8], and modify it to implement the proposed fault tolerance technique.

The OpenRISC 1200 is a 32-bit scalar, 5–stage integer pipeline RISC processor with Harvard microarchitecture. The control signals of the architecture were analyzed and it was determined that there were 53 control bits in total spread over 22 distinct control signals. Out of these, 44 are static in nature, while 9 are dynamic. Out of the 44 static signals, 25 of them are instruction dependent, while 19 are opcode dependent. Thus, for the considered processor, 47% of the control bits are protected by the static instruction dependent control caching scheme, while 33% are protected by the static opcode dependent caching scheme. The remaining 17% of the signals are protected by the dynamic control protection scheme. The proposed schemes were implemented on the RTL model and verified for functional correctness. FPGA synthesis was performed and the overheads analyzed. The details are presented in the following subsections.

6.4 Memory Overhead

The designed technique involves overheads in the form of extra storage required for the cache. These memory bits consume some power and can have an impact on the critical path of the design also. This subsection presents a detailed analysis of these metrics.

Considering the instruction dependent control cache first and denoting the number of control bits to be protected in

all the pipeline stages put together by N and the number of control cache entries by K, we can determine the extra amount of storage required as follows. The storage essentially is divided into three segments, the major contribution coming from the cache itself. The other components of the overhead are the storage requirements for the tag bits and the history bits.

The fact that a TMR structure is used for the control cache indicates that the number of memory elements in all the pipeline stages required for the control cache, M_{CC} is given by Equation 2, while the requirements for the history storage, M_{HS} is given by Equation 3.

$$M_{CC} = 2KN \tag{2}$$

$$M_{HS} = 2K \tag{3}$$

The number of memory elements required for the storage of the tag bits in the direct mapped structure, $M_{TB_{DM}}$ is given by Equation 4, while the number of memory elements required for the storage of the tag bits in the FIFO structure, $M_{TB_{FIFO}}$ is given by Equation 5.

$$M_{TB_{DM}} = K(30 - log_2 K) \tag{4}$$

$$M_{TB_{FIFO}} = 30K \tag{5}$$

Considering the opcode dependent control caching next and denoting the number of opcode dependent control signals by N_o and the number of distinct opcodes to index into the cache by K_o, the memory overhead without the ECC, M_{OCC} is given by Equation 6.

$$M_{OCC} = K_o N_o \tag{6}$$

The complete memory overhead, $M_{O_{DM}}$ for the direct mapped structure is given by Equation 7 and is the sum of the individual overheads

$$\begin{aligned} M_{O_{DM}} &= M_{CC} + M_{HS} + M_{TB_{DM}} + M_{OCC} \\ &= K(2(N+1) + 30 - log_2 K) + K_o N_o \end{aligned} \tag{7}$$

The complete memory overhead for the FIFO structure, $M_{O_{FIFO}}$ is given by Equation 8 and is the sum of the individual overheads.

$$\begin{aligned} M_{O_{FIFO}} &= M_{CC} + M_{HS} + M_{TB_{FIFO}} + M_{OCC} \\ &= K(2(N+1) + 30) + K_o N_o \end{aligned} \tag{8}$$

Analysis of the OpenRISC 1200 reveals that there are 53 distinct opcode combinations. Using Equation 7, it can be determined that a 512 entry direct mapped structure for control caching in the OpenRISC 1200 would require 37.5 kbits, while a 1024 entry cache based on the same configuration would take up 73 kbits. On the other hand, for a CAM based FIFO, Equation 8 suggests that a 512 entry structure would require 42 kbits of memory, while 1024 entries would require 83 kbits. In comparison to the huge sizes of modern processor caches, the area contributed by less than a hundred kilobits will not be significant.

6.5 Operating Frequency Overhead

The additional operations in the proposed scheme mostly take place in parallel with the already existing operations of the datapath. Analysis of the RTL implementation revealed that the addition of the history and tag tracking mechanism in the fetch stage had a penalty on the operating frequency. To counter this, it was decided to internally pipeline the fetch stage such that every instruction spends two cycles in it. The tag tracking and update were designed to be executed in two cycles. This affects the latency, but not the processor throughput.

In the case of instruction dependent control caching scheme and opcode dependent control caching scheme, the following delays are added to the path, if the protected control signals are needed in the beginning of a pipeline stage to proceed with the operations. Initially, the control cache needs to be accessed and the stored history read. The time for this is denoted by t_{CCA}. This data is input to the majority function module, which has at most two gate delays denoted by t_{MF}. There is a multiplexing delay of t_{Mux} involved in determining whether the output of the majority function module or the original signal itself is to be utilized. Thus, a possible overhead of t_{CCO} given by Equation 9 is added to the critical path timing.

$$t_{CCO} = t_{CCA} + t_{MF} + t_{Mux} \tag{9}$$

In most processors, these delays are not as significant as the delays involved in the rest of the operations performed in the pipeline stages. In the case that these delays severely hamper the operating frequency of the design, the control cache segment of each signal can be moved to the stage prior to the pipeline stage in which it is being utilized. Thus, the delays involved would be masked by the operations in the previous stage. However, the control signal is exposed to SEUs at the pipeline register storing the protected value.

6.6 FPGA Synthesis Results

Three different RTL models were subjected to synthesis on a Xilinx Virtex FPGA using the Xilinx ISE 7.1 synthesis tool. Table 1 summarizes the results.

The slice utilization goes up by 19% for the CAM based FIFO model and by 16% for the direct mapped model in comparison to the non-fault tolerant version of the processor. The flip-flop usage goes up by 10% for the CAM while it is only 3% more for the direct mapped structure. The penalty in operating frequency is just 6% in the direct mapped case, while it is as much as 21% in the CAM based FIFO case. It is due to the fact that the implementation of large sized CAMs in FPGAs are not efficient. High performance CAMs are possible in ASIC implementation. It appears that inspite of theoretically being superior, CAM

Table 1. FPGA Synthesis Results for OpenRISC 1200

RTL Model (Fault tolerance hardware)	Equivalent Gate Count	Operating frequency
OR1200 (No fault tolerance)	1,140,805	83.15 MHz
OR1200 (1024 entry CAM based FIFO)	1,610,570	66.02 MHz
OR1200 (1024 entry direct mapped)	1,544,827	78.07 MHz

based FIFO replaced control caches are not suitable for practical implementations in comparison to direct mapped structures.

7. Conclusions

A technique to protect the control signals of the datapath of a microprocessor has been proposed. It comprises of three integrated schemes, two of which involve the placement of a distributed cache to store the control signals of the instructions. A direct mapped structure for the instruction dependent control signals cache has been found to be most effective taking both the coverage and resource utilization into consideration. Fault coverage metrics for both 512 and 1024 cache entries have been determined, and the technique has been shown to protect 92% of all instruction executions with minimal area and cycle time overheads. A host of advantages are offered by the proposed technique in comparison with the previous related work [4]. The implementation of the technique is simple and there is a lesser overall overhead since penalty cycles are avoided due to the absence of the need for instruction re-execution in most cases. Recovery in case of detected faults also involves minimal cycles. The components of the scheme are also self-correcting. Possible future extensions to the work include the implementation of the above technique on multi-issue superscalar processors. Fabricating a processor along with the fault tolerance and detection mechanisms for the control logic and subjecting it to irradiation could validate the analysis presented here and provide practical proof of the efficiency of the proposed approach.

References

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *Micro 32: Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, November 1999.

[2] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical report 1342, University of Wisconsin, Madison. Computer Science Dept., 1997.

[3] E. Cota, F. Lima, and et al. Synthesis of an 8051-like microcontroller tolerant to transient faults. *Springer Science Journal of Electronic Testing*, 17(2):149–161, April 2001.

[4] T. S. Ganesh. Control caching: A fault-tolerant architecture for SEU mitigation in microprocessor control logic. Master's thesis, Iowa State University, 2006. Date Accessed: July 23, 2006. [Online]. Avaliable: http://archives.ece.iastate.edu/archive/00000229/.

[5] A. H. Johnston. Radiation effects in advanced microelectronics technologies. *IEEE Transactions On Nuclear Science*, 45(3):1339–1354, June 1998.

[6] S. Kim and A. K. Somani. On-line integrity monitoring of microprocessor control logic. *ICCD '01: Proceedings of the Nineteenth IEEE International Conference on Computer Design*, pages 314–321, September 2001.

[7] S. Kim and A. K. Somani. SSD: An affordable fault tolerant architecture for superscalar processors. *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, pages 27–34, December 2001.

[8] D. Lampret. Openrisc 1200 specification, opencores.org, 2006. Date Accessed: July 21, 2006. [Online]. Avaliable: http://www.opencores.org/cvsget.cgi/or1k/or1200/.

[9] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower. Ibm z990 soft error detection and recovery. *IEEE Transactions Device and Materials Reliability*, 5(3):419–427, September 2005.

[10] D. A. Patterson and J. L. Hennessy. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, third edition, 2002.

[11] M. Rebaudengo, M. S. Reorda, and M. Violante. Analysis of SEU effects in a pipelined processor. *IOLTW '02: Proceedings of The Eighth IEEE International On-Line Testing Workshop*, pages 112–116, July 2002.

[12] M. Rebaudengo, M. S. Reorda, M. Violante, P. Cheynet, B. Nicolescu, and R. Velazco. Coping with SEUs/SETs in microprocessors by means of low-cost solutions: A comparative study and experimental results. *IEEE Transactions On Nuclear Science*, 49(3):1491–1495, June 2002.

[13] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.

[14] N. Seifert, X. Zhu, and L. W. Massengill. Impact of scaling on soft-error rates in commercial microprocessors. *IEEE Transactions On Nuclear Science*, 49(6):3100–3106, December 2002.

[15] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *DSN '02: Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[16] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 30–39, July 2005.