Exploiting Existing Replicas of Stack Pointer in theRegister File for Error Detection

Exploiting Existing Replicas of Stack Pointer in the Register File for Error Detection

Hale Ogur, Davut D. Yavuz, Gozde Boztepe,
Serhat Gesoglu, Abdulaziz Eker, Oguz Ergin
TOBB University of Economics and Technology
Ankara, Turkey
Email: {hogur,st111101036,st111101024,
sgesoglu,aeker,oergin}@etu.edu.tr

Gulay Yalcin, Osman S. Unsal
Barcelona Supercomputing Center
Barcelona, Spain
Email: {gyalcin,ounsal}@bsc.es

***Abstract*—Hardware errors, i.e. Soft and Permanent Errors, became a tremendously important problem in the microarchitecture design as technology scales deeply and the number of transistors placed in computer systems increases exponentially. Moreover, single particle strikes may flip the values of several adjacent bits which causes multi-bit upsets. The register file is one of the most vulnerable structure in microprocessors since it is the major data holding component and referenced by most of the other components. Nevertheless, any access latency of register file will affect the rest of the pipeline which makes register file a time-critical component. Thus, it is essential to provide effective and fast reliability solutions to protect register files against hardware errors.**

In this study we present a fast error detection scheme in order to reduce the vulnerability of register file. We build our scheme on the observation that stack pointer is renamed repeatedly which causes keeping several copies of the same data in different registers. Thus, we leveraged these existing replicas for error detection. We show that our scheme can provide 59% reliability for the calculation of the Stack Pointer on average with a modest hardware overhead and with a negligible performance degradation.

I. Introduction

Technology trends are leading to more soft errors due to various phenomenons. For instance, scaling size of transistors increases the probability of multi-bit upsets due to high energy particle strikes. Also, utilizing lower voltages for energy savings increases the error rate. Although soft errors do not result in a permanent fault on the chip (and hence they are termed as soft errors), they may cause incorrect results or even a system crash if the error cannot be detected. The outcome of the fault generally depends on how reliability-critical the faulty component is. Register file is one of the most vulnerable structure in microprocessors since it is the major data holding component and referenced by most of the other components. Also within the register file, some registers are more vulnerable than the others. For instance, if the register is used for the address calculation, a fault in that register may lead to accessing incorrect address and may lead to system crash easily. Similarly, when the stack pointer is renamed in the physical register file, the register holding the value of the stack pointer becomes highly vulnerable since any fault in that register may cause accessing incorrect place in the stack and damaging all the stack related operations such as function returns etc.

It becomes increasingly essential to provide reliability solutions for computer systems. Error detection and error recovery are two major topics of the reliable computer architecture design. Redundancy is the key concept of error detection. Minimum two copies of the data or the execution is compared to see if there is any divergence due to a fault (Note that error is the manifestation of a fault.). Several redundancy-based error detection schemes execute instruction traces redundantly in different threads [11], [12], [14], or in different cores [2], [8], [13], [15] and compare the results of these redundant executions. However, these schemes present high energy overhead (i.e. around 100%) and double the resource utilization.

Another well-known error detection scheme for data structures is Error Correcting Codes (ECCs) in which each data value is extended with additional information (i.e. parity bits) for error detection. However, ECCs require additional encoding/decoding time which presents execution time overhead when ECC is utilized in the pipeline structures such as register file. In order to avoid delay overhead caused by ECC, the single-bit parity scheme can be used. However, it can only detect odd number of faults.

Reliability techniques introduce penalties in performance, power, die size, or design time. Therefore, it is essential to design simple and efficient reliability schemes. It is a technical challenge for researchers to develop simple (in terms of complexity) and cost-effective (in terms of less performance degradation and power consumption) error detection schemes. For instance, replication of the data and comparison to check the consistency of the replicated fields require additional hardware design and additional time during the execution. It has been shown that leveraging already existing structures in the processor together with the existing replication of data values provides efficient error detection [16]. The main challenge of this idea is to identify existing replication in a processor.

In this study, our goal is to reduce the vulnerability of register file by providing a low-cost error detection scheme by utilizing already existing replicas of the stack pointer. We build our scheme on our main observation that many stack operations are generally complementary (i.e. when the stack pointer is decremented by some amount, it is incremented by the same amount after a number of operations). Thus, stack pointer takes similar values during the execution of an application. Also, stack pointer is renamed to the physical register file. Therefore,

```
sub rsp(pr127),rsp(pr79 ),8
...
sub rsp(pr74 ),rsp(pr127),8
...
sub rsp(pr66 ),rsp(pr74 ),8
...
sub rsp(pr107),rsp(pr66 ),8
...
sub rsp(pr39 ),rsp(pr107),8
...
sub rsp(pr132),rsp(pr39 ),8
...
sub rsp(pr60 ),rsp(pr132),40
...
...
...
add rsp(pr53),rsp(pr60),40
...
add rsp(pr71),rsp(pr53),8
...
add rsp(pr57),rsp(pr71),8
...
add rsp(pr35),rsp(pr57),8
...
add rsp(pr42),rsp(pr35),8
...
add rsp(pr97),rsp(pr42),8
...
add rsp(pr137),rsp(pr97),8
```

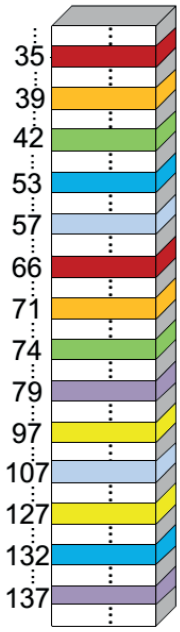

Fig. 1. Instruction trace from astar benchmark shown on the left and the physical register file status at the end of the trace on the right. Each instruction related to stack pointer (rsp) shows the allocated physical register number in parenthesis. In the end there are seven pairs of same values in the register file related to the stack pointer.

some significant portion of the physical register file is utilized for the stack pointer (We show it in Section. II).

In our scheme, we propose placing a small-sized Stack of Stack Pointer (SoSP) in the commit stage of the pipeline to keep the previous values of the stack pointer. So that, we utilize these previous values for a fast and low-cost error detection. (In Section. III, we present the details of our design.)

In our scheme, we detect errors before the faulty instruction commits. In this way, after detecting the error, for error-recovery we utilize already existing pipeline-flush mechanism of the pipeline. Thus, we provide a fast error detection without requiring the design of an additional error recovery scheme.

Our evaluations show that our scheme can reduce the vulnerability of the stack pointer calculation operations by 59% while reducing the vulnerability of the register file by 9% (Our experimental results are in Section. IV).

II. Motivation

Our design was motivated by the observation that many stack operations are complementary, i.e. when the stack pointer is decremented (or incremented) by some amount, it is incremented (or decremented) by the same amount after a number of operations. These operations cause the stack pointer to have the same values at different cycles. Due to renaming, which is common in modern microprocessors, the same values are held in the register file simultaneously, which creates some redundancy. To illustrate this point, Figure 1 shows a code portion taken from the SPEC benchmark *astar* on a 64 bit x86 processor. In this trace, the stack pointer is decremented

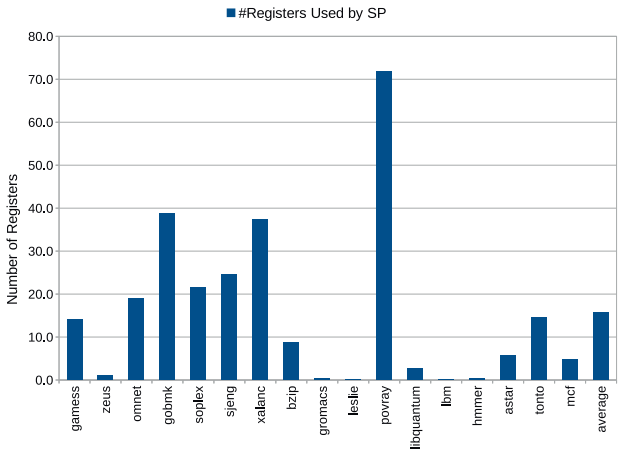

Fig. 2. Number of registers used by Stack Pointer on average during the execution time.

six times by 8 and once by 40. After some operations it is incremented once by 40 and six times by 8, restoring the context of several processor registers. In the end, multiple physical registers hold same values related to the stack pointer. Similar traces as Figure 1 can be seen in many programs.

We executed SPEC CPU2006 [3] applications in the MARSSx86 [9], full-system microarchitectural simulator and we found that the portion of the stack pointer related register file values constitutes up to 70% of the register file for some benchmarks and 15.6% of register file on average, as shown in Fig. 2. Here, the reason that the *povray* benchmark causes more stack pointer operations in close proximity, and thus constitutes a large portion in the register file is the way that POV-Ray algorithm works. Since POV-Ray algorithm performs a backward tracing, it needs to hold the traced data in memory. The processor uses stack memory heavily for this purpose. In contrast, the reason that *leslie* benchmark (based on LESlie3d computational fluid dynamics code) causes only a very small fraction of register file being related to the stack pointer is this benchmark mostly executes vector processing operations, which usually do not allocate the stack memory.

These statistics indicate that the reliability of the register file is significantly related to that of the stack pointer and protecting the stack pointer may reduce the vulnerability of the register file significantly.

The stack pointer operations that can be beneficial for soft error detection are usually due to calls to small subroutines, interrupt service routines that returns execution back to the same subroutine after the interrupt, and the code that explicitly pushes and pops entries in close proximity. However, some SP operations such as context switching may not be useful for this aim, where stack pointer is moved to a totally different memory address and works around that address for a very long interval.

III. Error Detection with Replicated Copies of SP

In this section, we first explain our design which implements a simple stack of SP (SoSP) for leveraging redundant values of stack pointer for reliability. We also present the extended version of SoSP which does not clear the popped

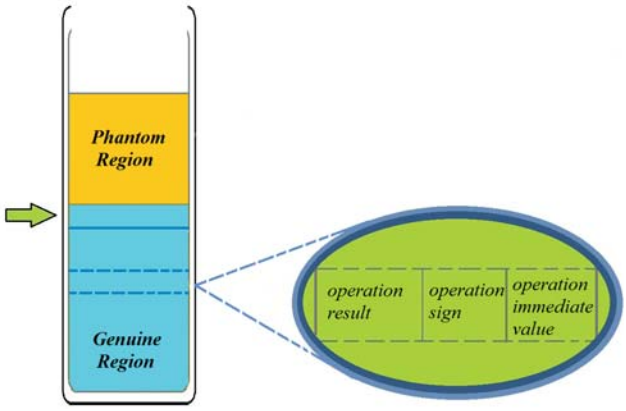

Fig. 3. The figure presents the organization of SoSP (Stack of Stack Pointer).

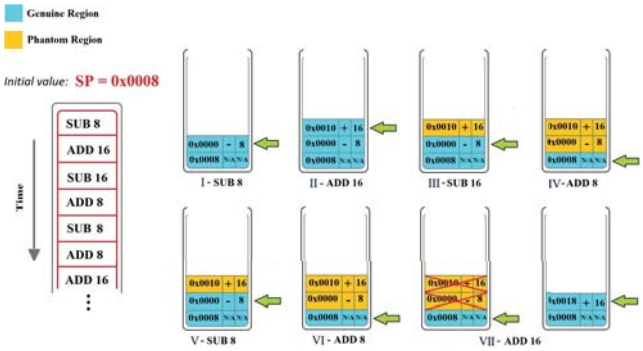

Fig. 4. The figure presents examples for the management of SoSP.

values of the SP from the stack, instead it continues keeping the value in the phantom area. Second, we present how the SoSP is managed. Third, we show the absract design of the required hardware.

A. Design

Based on the observation that generally stack operations are complementary, (if the stack pointer is decremented by some amount, it is incremented by the same amount after a number of operations.), we can detect a possible fault in the calculation of SP by using the previously calculated value of it. In order to do that, we introduce a Stack of Stack Pointer (SoSP) to the hardware. In the commit stage of the processor pipeline, when there is an operation performed on SP, we send the result, the immediate value and the operation to the SoSP. Note that, more than 90% of the operations performed on SP are either ADD or SUB operations.

If the operation is LOAD, it means the SP is initiating, thus, we flush the SoSP and push the result of the operation to it. If the operation is ADD (or SUB) there are two possibilities here. In the first possibility, it is the first time the value is calculated, thus, we push it to the top of the SoSP. In the second possibility, the value have already been calculated and is located at the top of SoSP. In this case, we use the saved value for error detection.

The key question here is how to determine if the calculated value is in the SoSP. For this reason, we check if the last operation performed on SP, which is also located at the top of the SoSP, is complementary to the operation committing now. For instance, the committing instruction can be increasing SP by 8 while the last operation performed on SP reduced its value by 8. In order to check if two operations are complementary, we compare the immediate values and operation types (i.e - or + operation). If the immediate values are equal while the operations are opposite it shows that the value is in the SoSP. In another word, committing instruction calculates the same value as the value of the SP before the last instruction in SoSP was executed.

When the calculated value is found in SoSP, in a naive approach, SoSP is managed as a normal stack, so that, the operation placed on the top of the SoSP is popped and deleted from SoSP. However, in order to increase the probability of finding the calculated value in SoSP, we continue keeping the operation in SoSP in the phantom area and we only update the pointer of SoSP to make it pointing to the top of the genuine area of SoSP. In order to detect if the calculated value is in the phantom area, after seeing that the value is not at the top of the genuine area, we also check the first entry in the phantom area. If both immediate values and operations are equal, it shows that the value is in the the phantom area of SoSP

In Figure 3, we show the main components of SoSP including the pointer, phantom area and genuine area of SoSP. Also, each entry of SoSP holds three required information of the executed instruction: the result of the operation, the immediate value (for ADD/SUB operations) and the operation type (is it ADD or SUB or other).

B. Management of SoSP

In Figure 4, we present an example for the management of SoSP. In the example, the initial value of SP is 0X008 and it is added to SoSP when the value is first loaded to SP. After the initial value, the first committed stack pointer instruction is SUB 8 (i.e. SP = SP-8) and we add it to the top of SoSP (case I). When ADD 16 arrives to commit stage as in case II, we first check if the operation is the opposite of the top of the stack and if the immediate values are the same. Since the condition does not hold (and also there is no other entry above), we push this instruction to the top of SoSP. In Case III, SUB 16 arrives. It has the same immediate value with the top of the stack and it has the opposite operation. In this case, we reduce the pointer of SoSP and compare the stored data with the result of the operation for error detection. Similarly, in Case IV, since the executed operation has the same immediate value with the top of SoSP while having opposite operations, again we reduce the pointer of SP and compare the stored data with the result. When SUB 8 arrives at case V, we see that it does not match with the top of the SoSP. We increase the pointer and see that there is a value stored in the pointed entry (in the phantom area). The pointed value has the same operation with the same immediate value, thus, we compare the data to detect errors. For case VI, the operation again matches with the top of the SoSP, thus, we reduce the pointer. Finally, in Case VII, the operation does not match the top of the SoSP or the first entry in the phantom area, thus, we write the new value to the top of the queue and then we flush the other entries in the phantom area above the written one.

C. Hardware Design

In Figure 5, we present the gate logic for our scheme. In the figure, the pointer of SoSP points to the top of the genuine region which is presented as SoSP[i]. Also, the instruction in the phantom region is shown as SoSP[i+1] while one instruction below the top of the SoSP is shown as SoSP[i-1]. The committing instruction is shown as WB_output.

In the figure, we first compare the immediate values and operation signs of SoSP[i] and WB_output (Note that in the figure, the operation signs are compared with a single AND gate while immediate values are compared with 64 AND gates in parallel.) If the immediate values are equal while the operations are opposite, it means that the new value of SP should be the same as the result in SoSP[i]. In this case, for error detection, we compare the result saved in SoSP[i-1] and the result of WB_output. In case of mismatch, we raise an error signal.

If the immediate value or the inverted operation of SoSP[i] do not match with the ones in the WB_output, we then check the phantom region. For this purpose, we first check if immediate value and operation sign of SoSP[i+1] are the same as in WB_output. If so, for error detection, we compare the results of SoSP[i+1] and WB_output. In case of a mismatch, we raise an error signal. In case, immediate values or the operation signs do not match, it means that the result of the committing instruction is not in the SoSP. In that case, we update SoSP[i+1] (the first instruction in the phantom region) with the WB_output and also we flush the other instructions above the updated one if there is any.

At the end of the error detection, we update the SoSP pointer either with SoSP[i-1] or SoSP[i+1].

IV. Evaluation

In this section we evaluate the reliability performance of our scheme and its overhead.

A. Simulation Environment

We evaluate our scheme by using MARSSx86 [9], full-system microarchitectural simulator. MARSSx86 uses X86 ISA, creates RISC-like microops and execute those microops in out-of-order pipeline. We used 18 benchmarks from the SPEC CPU2006 [3] using the reference input sets compiled for the x86-64 architecture with the optimization level O3. The simulated register file has 160 64-bit registers similar to the Intel Itanium Poulson, which is designed for mission critical servers. We simulated each benchmark for 100 million instruction commits after fast-forwarding for 1 billion instructions. We also set the size of SoSP as 16 entries in total for both phantom and genuine regions unless it is indicated otherwise.

B. Experimental Results

In order to evaluate the reliability performance of our scheme, first we measure how many times the calculated value of SP is also found in the Stack of SP (SoSP). For this experiment, we limit the size of SoSP with 16 entries to keep the hardware overhead minimum. The result of the experiment is presented in Figure 6. The observations from the figure are following. First, on average, the calculated value of the

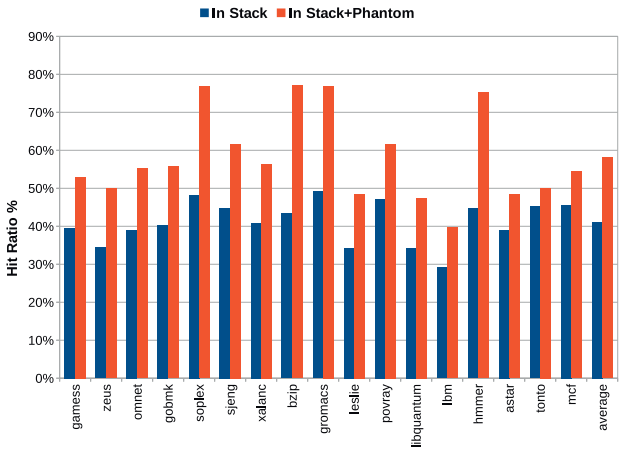

Fig. 6. The figure presents Hit Ratio: the ratio that the calculated SP value can be found in SoSP.

SP can be found in the genuine region for 40% of the time. Second, when we extend our design with the phantom area, the efficiency of our scheme increases to 59% on average. Thus, the vulnerability of the calculation of the stack pointer can be reduced from 100% (in the base case with no protection) to 41% when the SoSP is used with the phantom region extension.

In the Section II, we showed that 15% of the register file is occupied by some value of the stack pointer. Since we reduce the vulnerability of the SP by 59%, we also reduce the vulnerability of the register file by 9%.

In the next experiment, we evaluate the sensitivity of our scheme for the size of the SoSP. For this purpose, we increase the size of SoSP from 16 to 32, 64 and 256 entries and presented the results in Figure 7. Note that in the experiment we also utilized phantom region extension. The main observations from the figure are following. First, the hit ratio increases in most of the cases when the size of the SoSP is increased. Second, the increase of the hit ratio is not high enough to convince to use larger sized SoSP. Finally, in some cases increasing the size of the SoSP may reduce the rate of the hit. This is because, in some cases, complementary instructions may not come in the proper order. When the size of SoSP is small, at some point we had to flush it and make a fresh start. However, for a bigger sized SoSP, we flush it later and the disorganized SP calculations stay in SoSP longer.

The main overhead of our scheme is basically the 16-entry buffer with the size of only 260 bytes (16 entry$\times$(2 sign bits + 64 immediate bits + 64 SP value bits). According to our experiments, most of the immediate values can fit into 14 bits, thus 160 bytes is also enough for the buffer. Although this overhead seems high (i.e. equal size of having a register file with 20 entries), our scheme does not only detect the errors in the register file, it also detects errors in the pipeline stages from rename to commit during the calculation of SP. The error detection can be accomplished by 4 AND and 2 NOT gate passes which is less than a cycle and it can be done before the instruction commits. Thus, our scheme does not present a performance overhead. Compared to ECC or parity, our scheme can detect any number of multi-bit errors in one

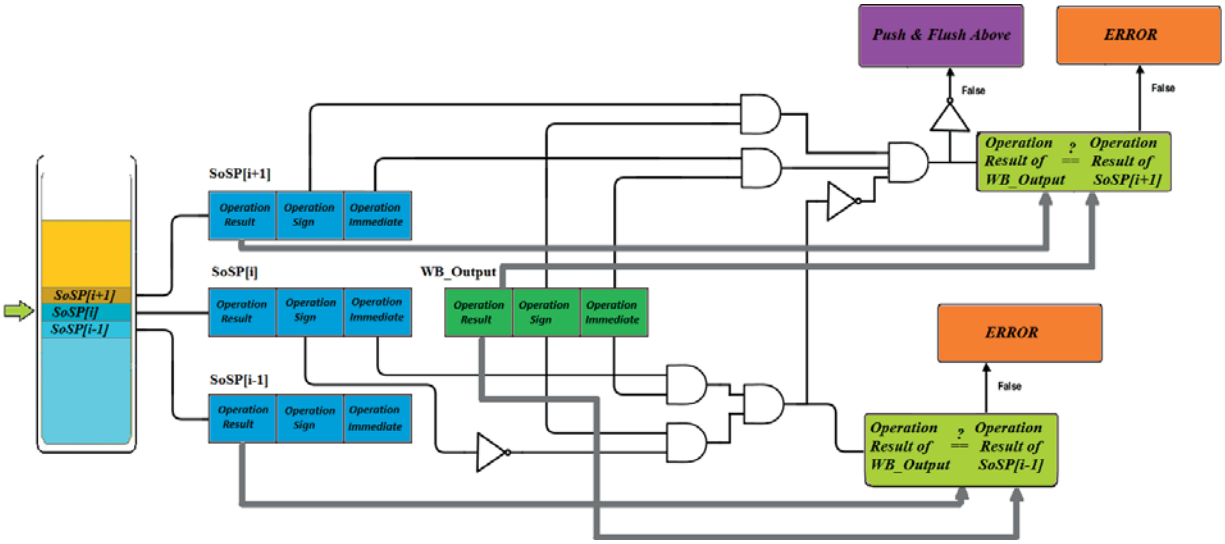

Fig. 5. The figure shows the hardware design of the proposed scheme for error detection and SoSP management.

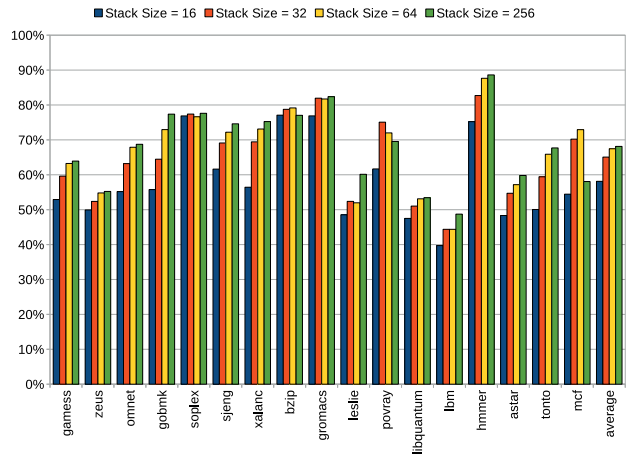

Fig. 7. The figure presents the Hit Ratio (the ratio that the calculated SP value can be found in SoSP) for different sizes of SoSP.

register or more than one registers renamed for stack pointer (i.e. similar to duplication). As we said before, compared to duplication, it can also detect errors occurred during the calculation of stack pointer without duplicating the pipeline stages.

V. Related Work

There have been several studies that try to increase the reliability of the register file. Pflanz et al. proposed using cross-parity check to detect multi-bit upsets in register files [10]. Cross-parity method relies on calculating three vectors for parities: row-parity vector, column-parity vector and diagonal-parity vector. Although this method can detect and correct up to 5-bit errors, it is costly to produce and check parity bits.

Kishani et al. presented Horizontal-Vertical-Diagonal (HVD) error detecting and correcting code which uses 4 parity vectors to protect data against soft errors [5]. While HVD can correct up to 3-bit faults in any position and combination, it requires more than 70% bit overhead.

Montesinos et al. proposed predicting the register lifetime to selectively protect registers by generating, storing and checking ECCs of the selected registers [7]. The proposed scheme is based on the idea that long-lived registers contribute AVF (architectural vulnerability factor) more.

Amrouch et al. observed that many number of bits are not used in the register file and they are set to 0 [1]. They proposed locating ECCs of the registers into their unused bits, so that, area and power overhead will be smaller while vulnerability is decreasing Karsli et al. also proposed [4] utilizing those unused upper part of the registers (i.e. narrow values) for duplicating the modified versions lower bits.

A similar scheme is also proposed by Memik et al. in which actively used registers are duplicated in unused physical registers [6]. Obviously, since the registers are highly used in the current applications, the scheme either presents a high performance overhead or it does not reduce the vulnerability enough.

Different than the previous approaches, our scheme can detect any number of faults occurred in the subset of the registers file (i.e. the subset which is used for stack pointer) while it can also detect errors in the execution path without duplicating the execution.

VI. Conclusion

In this study, we observed that Stack Pointer (SP) is calculated for the same values and the calculation operations are executed generally in the complementary order. Based on this observation, we saved the past values of SP in a stack, called Stack of Stack Pointer (SoSP) in order to leverage them for the error detection. Our results show that, we reduce the vulnerability of SP calculation by 59% which leads the reduction of the vulnerability of the register file by 9%.

ACKNOWLEDGMENT

This work was supported in part by TUBITAK under Grant 112E004. The work is in the framework of COST Action 1103.

REFERENCES

[1] Hussam Amrouch and Joerg Henkel. Self-immunity technique to improve register file integrity against soft errors. In *Proceedings of the 2011 24th International Conference on VLSI Design*, pages 189–194, 2011.

[2] Rui Gong, Kui Dai, and Zhiying Wang. Transient fault recovery on chip multiprocessor based on dual core redundancy and context saving. *International Conference for Young Computer Scientists*, pages 148–153, 2008.

[3] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34:1–17, 2006.

[4] Burak Karsli, Pedro Reviriego, M. Fatih Balli, Oguz Ergin, and Juan Antonio Maestro. Enhanced Duplication: A Technique to Correct Soft Errors in Narrow Values. *Computer Architecture Letters*, 12(1):13–16, 2013.

[5] Mostafa Kishani, Hamid R. Zarandi, Hossein Pedram, Alireza Tajary, Mohsen Raji, and Behnam Ghavami. Hvd: horizontal-vertical-diagonal error detecting and correcting code to protect against with soft errors. *Design Autom. for Emb. Sys.*, 15(3-4):289–310, 2011.

[6] Gokhan Memik, Mahmut T. Kandemir, and Ozcan Ozturk. Increasing register file immunity to transient errors. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, pages 586–591, 2005.

[7] Pablo Montesinos, Wei Liu, and Josep Torrellas. Using register lifetime predictions to protect register files against soft errors. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 286–296, 2007.

[8] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the International Symposium on Computer Architecture*, pages 99–110, 2002.

[9] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055, 2011.

[10] Matthias Pflanz. *On-line Error Detection and Fast Recover Techniques for Dependable Embedded Processors*. Springer-Verlag, 2002.

[11] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Computer Architecture News*, 28(2):25–36, 2000.

[12] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, page 84, 1999.

[13] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19:12–23, 1999.

[14] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture*, pages 87–98, 2002.

[15] Alan Wood, Robert Jardine, and Wendy Bartlett. Data integrity in HP NonStop servers. In *Proceedings of the Workshop on System Effects of Logic Soft Errors*, 2006.

[16] Gulay Yalcin, Oguz Ergin, Emrah Islek, Osman Sabri Unsal, and Adrián Cristal. Exploiting existing comparators for fine-grained low-cost error detection. *Transactions on Computer Architecture and Code Optimization*, 11(3):32:1–32:24, 2014.