International Journal of Computer Applications (0975 – 8887)
Volume 92 – No.7, April 2014

Dynamic Resources for Multicore Processor using Register File Protection

H. G. Konsowa
Faculty of Engineering, Helwan University, Cairo, Egypt

E. M. Saad
Faculty of Engineering, Helwan University, Cairo, Egypt

M. H. A. Awadalla
Faculty of Engineering, Helwan University, Cairo, Egypt
Electrical and Computer Engineering Dept., SQU, Oman

ABSTRACT
A massive investment in the design multicore has been accomplished through technologies that impose significant barriers to assure the reliable operation of future chips. Extremely complex, parallel, multi-core processor fabricated in these technologies will become more vulnerable to several factors that produce transient (soft) errors or permanent (hard) errors. One of the critical issues to protect a processor is the register file. It registers the architectural states for long periods and also it can be read frequently. This paper presents a new eviction policy to the registers entry from error code correction table in the insertion stage for the integer register file protection process. The paper presents a qualitative comparison with other eviction policies (random and the least recently used, LRU). Also it addresses the effect of using the integer register protection with dynamic resource fetch policy on the overall performance by adding the protection for integer registers files to the dynamic allocated resource (fetch policy). The achieved results show that the dynamic fetch policy WZ-FETCH outperforms in all addressed benchmark programs in case of using register file protection.

General Terms
Multicore Processor Design, Dynamic design

Keywords
Multicore, Resource allocation, Register sharing, Register renaming, Simultaneous Multi-Threading (SMT).

1. INTRODUCTION
Simultaneous Multithreading (SMT) increases processor throughput by permitting the parallel execution of many threads. Static resource sectionalization techniques have been suggested, but are not as effective as dynamically controlling the resource usage of each thread since program phases are not fixed all the time. Static resource partitioning [1], [2] evenly split up critical resources among all threads, thus preventing resource monopolization by a single thread. However, this method lacks flexibility and can cause resource to remain idle when one thread has no need for them, even if other threads could benefit from additional resources. This section includes a briefly review for previous works on dynamic resource allocation in multiprocessor, multithreaded and multicore platforms. Although several proposals that address the management of a single micro-architectural resource exist in the literature, proposals to manage multiple interacting resources on multicore chips at runtime are much scarce. The authors in [3] proposed an algorithm that dynamically assigns resources to each thread according to thread behavior changes. Advanced Real-time Processor Architecture (ARPA) system analyzes the resource usage efficiency of each thread in a time period and assigns more resources to threads which can use them in a more efficient way. The purpose of ARPA is to improve the efficiency of resource utilization, thereby improving overall instruction throughput. In microprocessor organization' arena as microarchitectural complexity increases, (crossing instruction-layer correspondence to thread level-parallelism and toward multi-core and many-core architectures), it is more difficult to explain concepts like cache, out-of-Order and speculative execution, power consumption, and the fundamental interaction among the architecture components. It is important to know the architecture concepts by observing the flow of instructions in time, also by exploring the impact of different processors configuration on performance. Any simulator must exhibit both the structural relationships between microarchitectural components and the temporal dependences between executed instructions that are in-flight in the pipeline structures. Multicore processor architectures incorporate CPU cores, memory arrays (e.g. caches, register files), memory control logic and interconnection logic. The register files are one of the critical structures of these memory arrays to protect a processor. It is a sizable structure that stores architecture state. It often stores data for long periods and is read frequently, which increases the probability of spreading a faulty datum to other parts of the machine. The register files that occupy a large portion of processor die can be successfully protected using well-known information-redundancy techniques like Error-Correcting Codes (ECC). Thus, the key element of online error detection is to protect the units of the processor, CPU cores (which dominate the remaining die area), the memory hierarchy control logic (memory consistency), and the interconnection logic. Several online error detection techniques for the register files have been recently proposed .A protection mechanism for soft errors in register files should have no performance shock; keep the remaining Architectural Vulnerability Factor (AVF) [4] to a small value. To implement such a mechanism, the two observation parameters on the use of registers in general-purpose processors are observed. The first one is that the data stored in a physical register is not always useful. Not all soft error in a physical register will affect the processor's architectural state. Consequently, the register needs to be protected when it is contained a useful data. The second observation is that not all the registers are equally vulnerable to soft errors. A small set of long-lived registers account for a large fraction of the time that registers need to be protected. The contribution of the most other registers to the vulnerable time could be ignored. The paper is organized as follows. Section 2 describes the related work to this paper. Section 3 illustrates the proposed methodologies. Section 4 describes and discusses the simulation results. Section 5 concludes the paper.

2. RELATED WORKS

The background will cover three related areas:

- Register life time analysis and simulation framework.
- Microarchitecture level soft error vulnerability analysis.
- Dynamic fetch policies.

2.1 Register Life Time Analysis and Simulation Framework

In recent out-of-order processors, the instructions are fetched, decoded, and then sent to the rename stage. After the processor decodes an instruction with a destination register, it allocates a free physical register, creates a new register version. Instructions in the Issue queue until they are selected for carrying out by the Select stage. Select stage selects an instruction for execution once all of the source operand is ready, and the instruction is at the head among the ready instructions. Later, the instruction is executed in a Functional unit. After execution, an instruction's solvent is broadcasted on the Bypass network, so that any dependent instruction can immediately use it. The result is also written to the corresponding physical register, and the instruction updates its ROB status. The instruction retires once it reaches the head of the ROB.

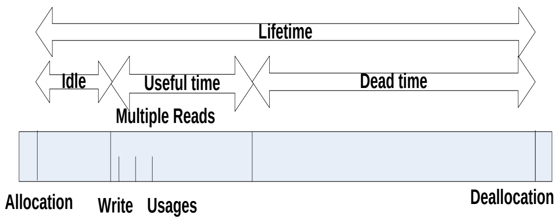

Fig 1: Physical register life cycle

To explain the rename stage, the register version is kept until the instruction that determine the corresponding logical register retires, this is necessary to handle precise exceptions. Observe that a register version is written to only once but can be read multiple times. . As Fig. 1 shows, the life-time of a register version lasts from register allocation to de-allocation. This full period is divided into three different intervals: allocation until write interval; write until last read interval; last read to de-allocation interval. These intervals are called PreWrite, Useful, and PostLastRead respectively. This means that only the Useful period time needs to be protected. The used simulation (Multi2Sim) is modified [5] to be capable of simulating dynamic resources for multicore. This simulation is a framework for heterogeneous computing systems, including models for super-scalar, multithreaded, multicore, and graphics processors. Multi2Sim simulator is adapted to cope with multicore processor dynamic design by adding dynamic feature in the policy of thread selection in fetch stage [6].

The used framework consists of multicore simulation tool and a subset of benchmark programs used to evaluate an architectural enhancement of multicore by workload all threads of multicore using one benchmark program which can be executed in parallel way or multiple benchmark programs that can be executed in sequential workload way [7]. Mult2sim simulation tool supports a set of parameters that specify how stages are organized in multithreaded design. The pipeline model in multicore simulator is divided into five stages as shown in Fig. 2: fetch stage, decode stage, rename stage, issue stage, execution stage and commit stage. Stages can be shared among threads or private per thread except execute stage, which is shared by definition of multithread.

The register renaming mechanism is implemented in simulation. Multi2sim uses a simplification of x86 logical registers. There are 32 possible logical dependences between microinstructions, which are listed in Fig.3. a.

- General purpose registers are used for computations and intermediate results.
- Specific purpose registers implicitly or explicitly modified by some microinstructions, such as the stack pointer or base pointer for array accesses.
- Segment registers.
- Multi2sim specific register are internally used by the macroinstruction decoder to communicate corresponding microinstructions with one another.

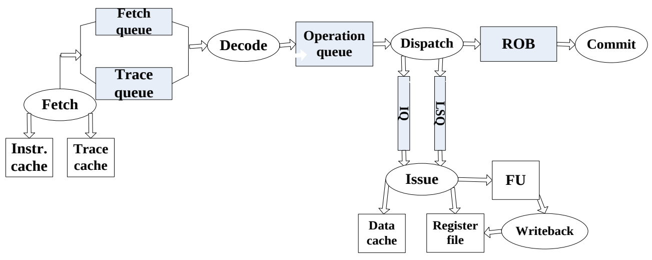

Fig 2: Processor pipelines.

The x86 architecture uses a set of flags that are modified by some arithmetic instructions, and later consumed mainly by conditional branches to decide whether to change the program sequence or not. Flags such as of, cf, and df are overflow, carry, and direction flags respectively and they are tracked as separate dependences among instruction. On the other hand, flags zf, pf, and sf are zero, parity, and sign flags respectively, these three flags are named zps as shown in Fig. 3 a and any x86 instruction can modify all of them. Thus they are tracked as a single dependence. The value associated with each logical register, i.e. each potential input dependence for an instruction is stored in the physical register file. As represented in Fig.3 b, the register file consists of a set of physical registers that store operation results. Each physical register is formed of a 32-bit data, jointly with a 6-bit field storing the x86 flags. The implementation of rename process into the multi2sim simulation is explained as follows: at a given instant, each logical register is mapped to a given physical register in the register file, containing the associated value. In used renaming model, logical register and renaming work independently. This means, for example that register eax and flag cf can be mapped to the same register file entry. In this case, the value field stores the contents of eax, while a specific bit in the flags field contains the value for cf. Each logical register is mapped to a different physical register, but x86 flags can be mapped all to the same physical register, even if the latter already has an associated logical register.

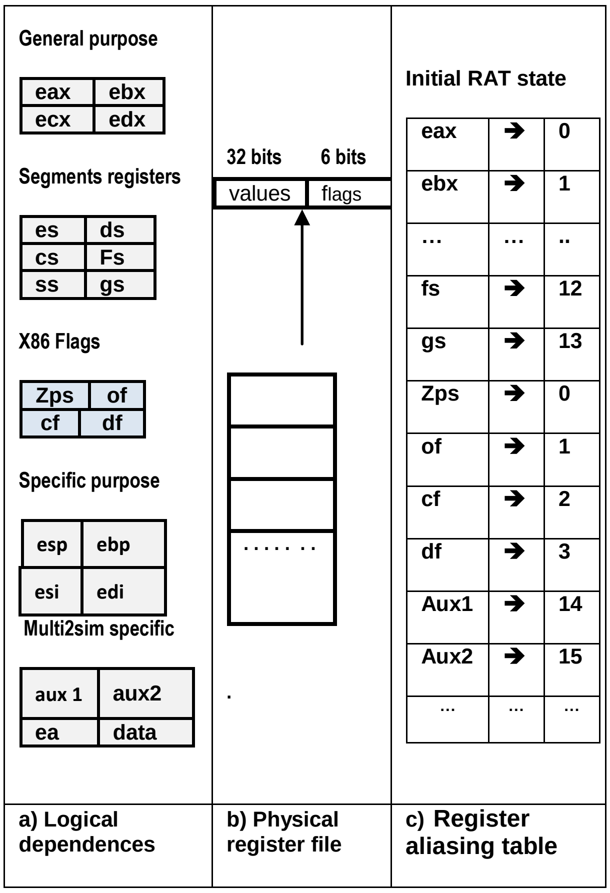

Fig 3: Register renaming.

A Register Aliasing Table (RAT) holds the current mappings for each logical register. Additionally, a Free Register Queue (FRQ) contains identifiers corresponding to free (not allocated) physical registers. When new instruction writing into logical register i is renamed, a new physical register is taken from FRQ and the new mapping for i is stored in RAT. The previous mapping p0 of logical register i will be needed later, and is stored in the ROB entry associated with the renamed instruction. When subsequent instructions consuming i are renamed, RAT will make them read its contents in p, when they will find the associated value

When the instruction writing on i is committed, it releases the previous mapping of i, i.e., physical register p0, returning it to FRQ if necessary. Notice that, unlike a classical renaming implementation ignoring flags, a physical register can have several entries in RAT pointing to it (the maximum is the number of flags plus one logical register). Thus, a counter is associated with each physical register, which will only be freed and sent back to FRQ in case of this counter is 0.

2.2 Microarchitecture Level Soft Error Vulnerability Analysis

The microarchitecture level, broadcast vulnerability to soft errors can be modeled using several methodologies. For example, Li and Adve [8] estimate the reliability using a probabilistic model of the error generation and propagation process in a central processing unit In the past, statistical fault injection has also been used in several studies [9]-[11] to evaluate architectural reliability. In this work, the reliability of central processor microarchitecture structures is estimated using the Architectural Vulnerability Factor (AVF) computing method acting introduced in [12], [13].

AVF of a hardware system is the probability that a transient fault in that hardware structure can cause erroneous output of a program. The overall hardware structure's computer error rate is determined by two factors: the raw error rate of the hardware device, mainly determined by circuit design and processing technology, and AVF. Since the hardware raw error rate normally does not vary with code execution of instrument, AVF can be used as a good dependability figurer to quantify how vulnerable the hardware is to soft errors at different program execution phase. To compute AVF, it needs to distinguish those fleck that affect the final arrangement output and those that do not. A subset of processor state of bits required for architecturally correct execution (ACE) are called ACE bits [12]. AVF of a hardware structure in a given cycle is the percentage of ACE bits that the structure holds, and AVF of a hardware structure during program execution is the average AVF at any point of time. In practice, identifying un-ACE bits, the processor state bits that do not affect correct program execution, is much easier. Examples of locations that often contain un-ACE bits include idle or invalid states, uncommitted instructions and dynamically dead instructions and data. An instruction is considered dynamically dead if its result is not used by any other instructions and therefore will not affect the final output of the program. A cycle accurate execution driven simulator can be used to identify un-ACE bits and to track the residency cycles of the un-ACE bits in hardware structures. To compute AVF of the entire processor, AVF of all hardware components are added together after being weighted with the number of bits in each structure.

P. Montesinos et al. [14] proposed a ParShield, as a novel architecture that provides cost-effective protection for registry file against soft errors. ParShield relies on the Shield concept, which selectively protects a subset of the registers by generating, storing, and checking ECC s of only the most vulnerable registers while they contain useful information. Shield supports three operations on one such register: (i) when the register is written, Shield generates and save ECC of the written data, (ii) when the register is read , Shield checks whether the register contents are still valid, and (3) Shield keeps ECC of the data until the register is read for the last time. Shield assumes a single -bit fault model.

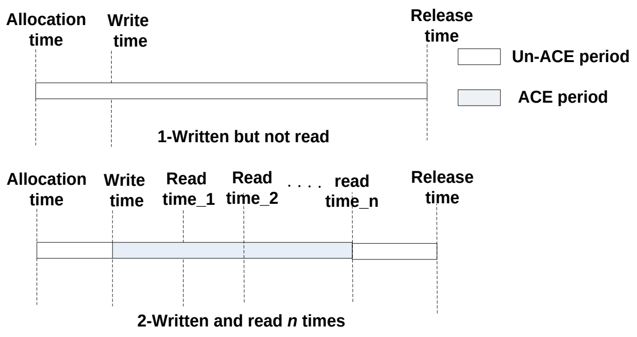

Fig 4: ACE periods for two register versions.

Mukherjee et al. [12] proposed the concept of Architecturally Correct Execution (ACE) to compute a die structure 's AVF. ACE analysis divides a bit's lifetime into ACE and un-ACE periods. A bit is in ACE state when a change in its value will produce an error AVF for a one bit is the fraction of time that it is in ACE state. To calculate the amount time a bit in ACE State, the first assumption is the whole register life time is in

ACE state, and then the fraction that can be proven as it is un-ACE state will be removed. The fraction left is an upper leap on the ACE time.

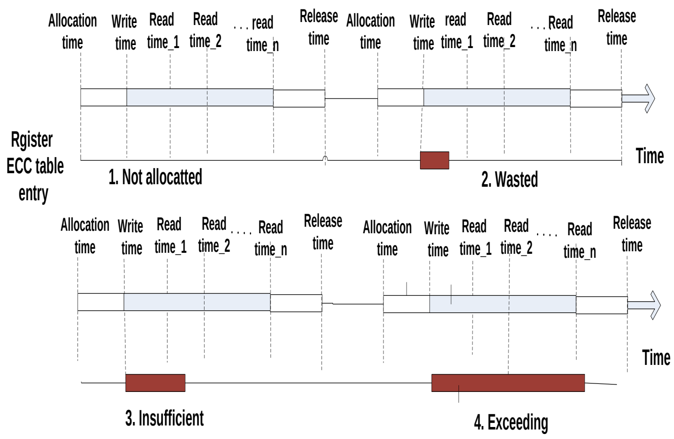

Fig 5: Physical register life cycle.

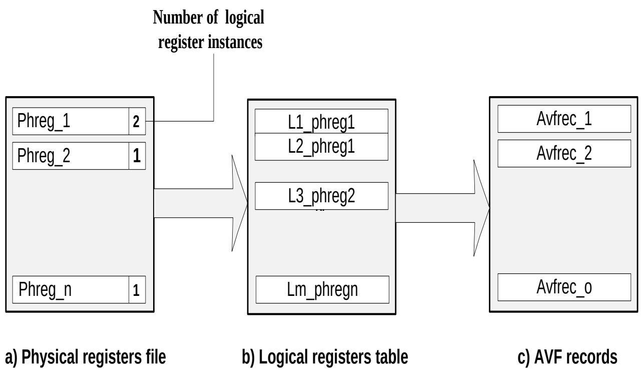

Fig 6: Registers lifetime framework

2.3 Fetch Policies

Static fetch policy is used to be compared with dynamic fetch policy (WZ-FETCH) which is introduced in [15]. The static fetch policy selects a certain thread to serve for certain time slice then switch to another thread and so on but WZ-FETCH policy depends on the Ordinary Least Square (OLS) regression statistic method [16]. WZ-FETCH policy, the used algorithm of selecting fetch threads, will select the thread which has least miss value of L2 data cache miss to increase data locality. To predict the future L2 data cache miss, OLS regression equation using number of samples equal to the window array size for each thread will be used, i.e. the function to calculate the future L2 data cache miss is found on thread level. This function is called regression engine. The WZ-FETCH fetch policy is the best fetch policy in all used benchmarks programs for all used metrics.

WZ-FETCH fetch policy is represented as a history-aware resource because it used previous data to take decision based on prediction information.

This paper is focused on characterizing AVF phase behavior of an individual microarchitecture structure where the study of a component based reliability analysis is more suitable for the design and optimization of reliability-aware architecture. For example, the hardware components that yield high AVFs can be identified and protected, either at the design or run-time stages.

3. THE PROPOSED METHODOLOGY

This paper is focused on three issues:

- Register life time analysis.
- Register file protection with new eviction police.
- Comparison between different fetch policies using register file protection.

3.1 The Register Life Time Analysis

The register life time analysis includes the following:

- A study for a used register type for some splash benchmarks weather integer or float measured by the total consumed cycles from allocation to de-allocation period.
- Breakdown of useful periods for integer registers for some of splash benchmarks.
- A study for all integer and float registers to compare between useful and post last read periods. for register life time.

3.2 Register File Protection

The register protection study states that the protection overhead for Barns benchmark as an example using processor design consists of two cores, each of them includes two threads. The new contribution in this part is adding new eviction policy to the registers entry from ECC table when it is fully occupied. Also the effect of using different eviction replacement policy for register protection will be measured. This can be accomplished by comparing the number of reads of register aliasing table in case of using register life time protection and no protection used. When using protection register life time, two eviction replacement polices are used, random and least recent used (LRU) policies and compared with the developed strategy in (LRU99). This work is applied using Barnes, FFT, FMM and Sort Benchmarks. There is no problem to add new entry in ECC table if it is empty. However, if it is full so any used entry can be freed by applying random or LRU eviction policy. In the new eviction policy, it uses the same concept of LRU but it is different in the starting of searching window. This searching can be started from the current location (LRU 99) for time saving.

3.3 Comparison between Different Fetch Policies Using Register File Protection

In this section, the comparison between static fetch and dynamic fetch policy (WZ-FETCH) is introduced in [12]. The static fetch policy selects a certain thread to serve for certain time slice and then switch to another thread and so on however WZ-FETCH policy depends on the Ordinary Least Square (OLS) regression statistic method [13]. To predict the future L2 data cache miss, OLS regression equation, called regression engine, using number of samples equal to the window array size for each thread is presented. The used algorithm of selecting fetch threads will select the thread which has least miss value of L2 data cache to increase data locality. WZ-FETCH is the best fetch policy in all used benchmarks programs for all used metrics.

4. SIMULATION RESULTS

For register life time analysis, Instruction Per Cycle (IPC) is the main measurement unit in this paper. IPC throughput is measured as the sum of the IPC values of all running threads, it measures how effectively resources are being used. It is necessary to select the number of consumed CPU cycles to be used to differentiate between short life time and long life time registers. Fig. 7 and Fig. 8 are described that if the consumed cycles is less than or equal 10 cycles, the register is defined as short life time register otherwise the register is defined as long life time register.

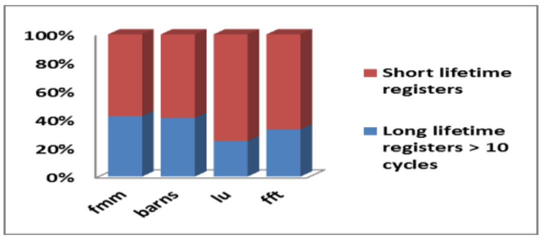

Fig 7: Integer registers life time analysis for some SPLASH benchmark programs. .

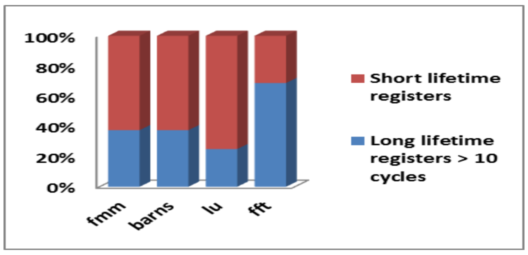

Fig 8: Float registers life time analysis for some SPLASH benchmark programs.

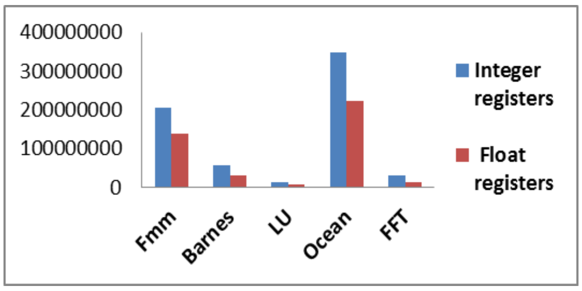

Fig. 9: Total consumed CPU cycles from allocation to de-allocation periods for integer and float registers.

Total consumed CPU cycles from allocation to de-allocation periods for integer registers is greater than consumed cycles for float registers as shown in Fig. 9. So, all incoming analysis will be based on integer registers.

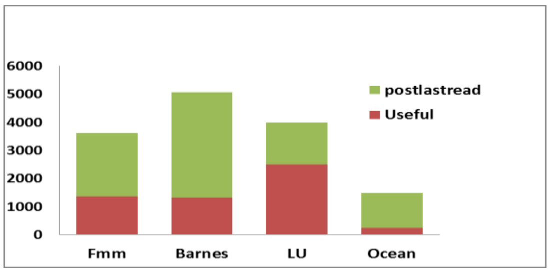

Fig 10: Average consumed cycles by integer registers.

Fig.10 illustrates that the useful period for all integer registers is a small fraction of the register's lifetime.

Fig. 11 shows the useful period breakdown for integer registers. This period is divided to writes, reads and de-allocations periods for each register

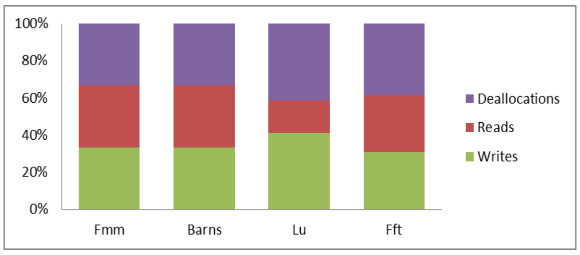

Fig 11: Cycles breakdown of useful period for integer registers for some of Splash benchmarks.

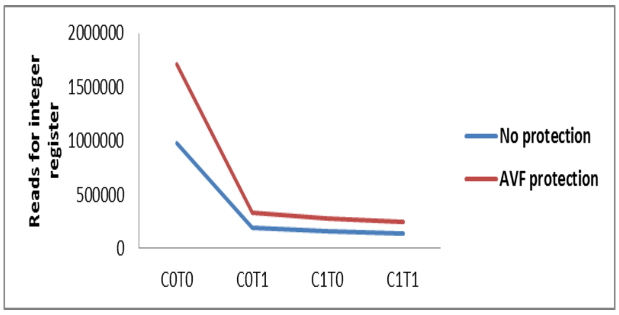

Fig 12: Register protection overheads for BARNES benchmarks from Splash benchmarks.

Fig. 12 shows the protection overhead for Barnes benchmark as an example using processor design consists of two cores each of them includes two threads.

Fig. 13 states that the number of CPU cycles which are consumed when accessing of AVF hardware component. This AVF hardware component is used for protection the useful life time of integer registers. Some of benchmarks from Splash suite are used for this figure.

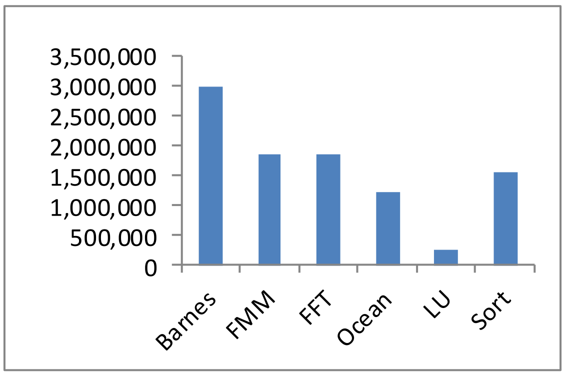

Fig 13: Consumed CPU cycles for protection of integer register file (AVF).

Fig.14 presents the number of reads in ECC tables for several benchmarks from Splash suite which represents the protection latency. To draw this figure, read counter is added to count all reads from ECC table to check the register value correctness.

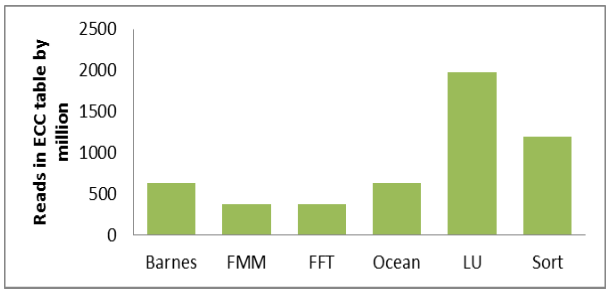

Fig 14: Number of reads in ECC tables (AVF Latency).

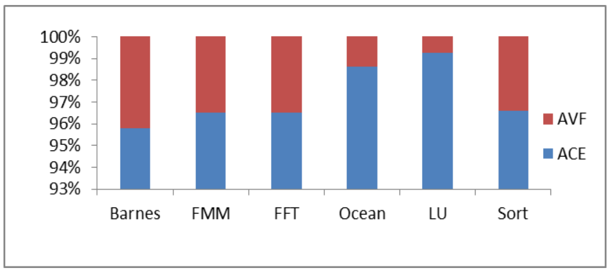

Fig 15: Register life time analysis after adding protection feature measured by distribution percentage of consumed CPU cycles.

The addition of ACE to the multicore design leads to more consumed CPU cycles. AVF takes part of these cycles. This figure affirms that AVF for Barnes benchmark consumed round 4% from ACE consumed CPU cycles and 1% for LU benchmarks using ECC tables with limited size to store the register values from first read to de-allocation period which is the critical values of the integer registers. After many trials, ECC size is selected to be 100 entries for that ECC table. In this section, the results of applying protection using many eviction or replacement policies or no protection on many benchmarks are illustrated in Fig. 16 to Fig. 19, these ECC eviction policies are Random, LRU or LRU99, the result of LRU99 outperforms.

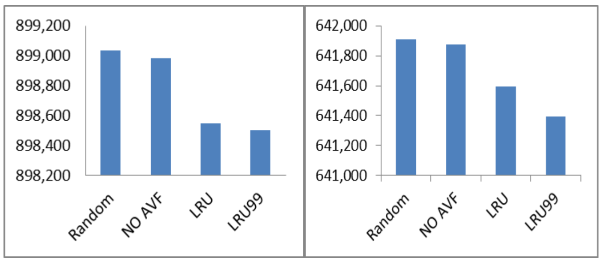

Fig 16: Number of access for register aliasing table in Barnes benchmark.

(a) Reads. (b) Writes.

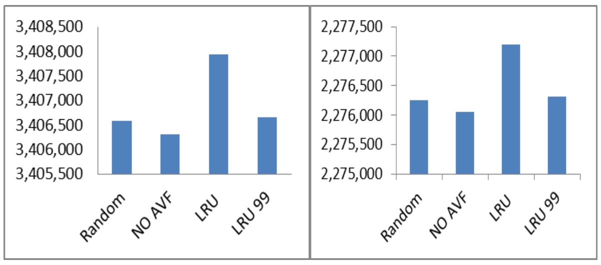

Fig 17: Number of accesses for register aliasing table in FMM benchmark.
(a) Reads. (b) Writes.

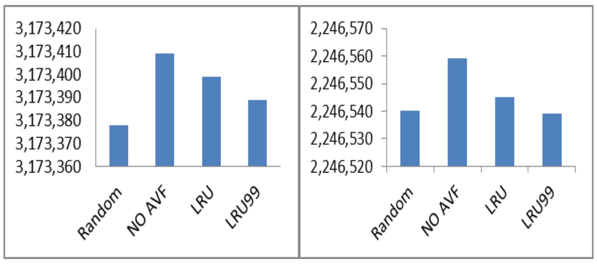

Fig 18: Number of accesses for register aliasing table in FFT benchmark.
(a) Reads. (b) Writes

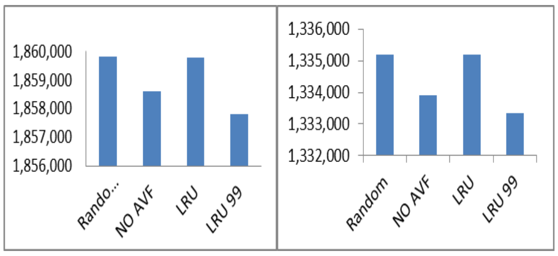

Fig 19: Number of accesses for register aliasing table in Sort benchmark.
(a) Reads. (b) Writes.

Moreover, different fetch policies “Static and WZ_FETCH” are applied on several benchmarks of SPLASH suite after adding AVF protection for the register files. The protection is done using LRU99 eviction policy. Fig. 20 depicts the number of CPU cycles which consumed in reading of protected integer registers for all used benchmarks. The WZ-FETCH fetch policy is the best fetch policy in all used benchmarks programs except in LU benchmark. The CPU read cycles is the same value for the two fetch policies. Fig 21 shows the average CPU cycles consumed by integer register for all the life time periods from allocation to de-allocation for many benchmarks using different fetch policy. It is proved also that WZ_FETCH fetch policy is the best fetch policy. This means that the use of register file protection does not affect the ranking of WZ_FETCH fetch policy as the best fetch policy.

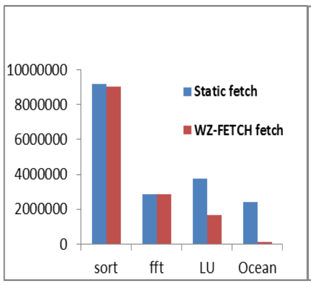

Fig 20: Average CPU cycles consumed by integer registers between allocations to de-allocations for different fetch policies.

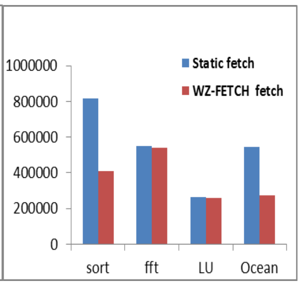

Fig 21: Number of CPU read cycles for integer registers using AVF protection for different fetch policies

5. CONCLUSIONS

In this paper, the proposed mechanism relies on using dynamic resource in the fetch stage with register protection, which protects a subset of the registers by generating, storing, and checking the ECCs of only the most vulnerable registers. The paper presents an eviction policy to the registers entry from ECC table in the insertion stage for the integer register file protection process and compares it with old eviction policies (random and LRU). Also it addresses the effect of using the integer register protection with dynamic resource fetch policy on the overall performance by adding the protection for integer registers files to the dynamic allocated resource (fetch policy). The achieved results depicts that the dynamic fetch policy WZ-FETCH outperforms in all addressed benchmark programs in case of using register file protection. This means that the use of register file protection does not affect the ranking of WZ_FETCH fetch policy as being a best fetch policy.

6. REFERENCES

[1] Marr, D. T., Binns, F., Hill, D. L., G. Hinton, Koufaty, D. A., Miller, J. A. and Upton, M. 2002. Hyper-Threading Technology Architecture and Microarchi-tecture,. Intel Technology J., vol.6, no.1, (Feb. 2002), 4-15.

[2] Raasch, S. E. and Reinhardt, S. K. 2003. The Impact of Resource Partitioning on SMT Processors. Proc. 12th Int'l Conf. Parallel Architecture and Compilation Techniques (Sept. 2003), 15-26.

[3] Wang, H., Koren, I. and Krishna, C. 2011. An Adaptive Resource Partitioning Algorithm in SMT Processors. Parallel and Distributed Systems, IEEE Transactions on, Volume: 22, Issue: 7 July.

[4] Slegel, T., Averill, I., R.M., Check, M., B. Giamei, Krumm, B., Krygowski, C., Li, W., Liptay, J., MacDougall, J., McPherson, T., Navarro, J., Schwarz, E., Shum, K. and Webb, C. 1999. IBM's S/390 G5 Microprocessor Design. IEEE Micro, vol. 19.

[5] "The Multi2Sim Simulation Framework" http://www.multi2sim.org, 2011.

[6] Konsowa, H. G., Saad, E. M. and Awadalla, M. H. A. 2012. Updating Multicore Processor Simulator to Support Dynamic Design in Fetch stage. National Radio Science Conference (NRSC).

[7] Ubal, R., Sahuquillo, J., Petit, S, and L_opez, P. 2012. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing.

[8] Li, X. D., Adve, S. V., Bose, P., and Rivers, J. A. 2005. SoftArch: An Architecture Level Tool for Modeling and Analyzing Soft Errors, In Proceedings of the International Conference on Dependable Systems and Networks.

[9] Kim, S., and Somani, A. K. 2002. Soft Error Sensitivity Characterization of Microprocessor Dependability Enhancement Strategy. The International Conference on Dependable System and Networks.

[10] Wang, N. J., Quek, J., Rafacz, T. M. and Patel, S. J. 2004. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline, In Proceedings of the International Conference on Dependable Systems and Networks.

[11] Czeck, E. W. and Siewiorek, D. 1990. Effects of Transient Gate-level Faults on Program Behavior, In Proceedings of the International Symposium on Fault-Tolerant Computing.

[12] Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K. and Austin, T. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor, In Proceedings of the International Symposium on Microarchitecture.

[13] Biswas, A., Cheveresan, R., Emer, J., Mukherjee, S. S., Racunas, P. B. and Rangan, R. 2005. Computing Architectural Vulnerability Factors for Address-Based Structures, In Proceedings of the International Symposium on Computer Architecture.

[14] Montesinos, P., Liu, W. and Torrellas, J. 2007. Using Register Lifetime Predictions to Protect Register Files against Soft Errors. Dependable and Secure Computing (TDSC), IEEE Transactions on, Volume: 22, Issue: 7 (June 2007).

[15] Konsowa, H. G., Saad, E. M. and Awadalla, M. H. A. 2012. New Fetch Policies for Multicore Processor Simulator to Support Dynamic Design in Fetch Stage. JOURNAL OF COMPUTER SCIENCE.

[16] Cottrell, A. Regression Analysis: Basic Concepts. [Online]. Available: Regression.pd.