Increasing Register File Immunity to Transient Errors*

Gokhan Memik[1], Mahmut T. Kandemir[2], Ozcan Ozturk[2]

[1]Electrical and Computer Engineering Dept.
Northwestern University
memik@ece.northwestern.edu

[2]Computer Science and Engineering Dept.
Pennsylvania State University
{kandemir, ozturk}@cse.psu.edu

ABSTRACT

Transient errors are one of the major reasons for system downtime in many systems. While prior research has mainly focused on the impact of transient errors on datapath, caches and main memories, the register file has largely been neglected. Since the register file is accessed very frequently, the probability of transient errors is high. In addition, errors in it can quickly spread to different parts of the system, and cause application crash or silent data corruption.

This paper addresses the reliability of register files in superscalar processors. Particularly, we propose to duplicate actively used physical registers in unused physical registers. The rationale behind this idea is that if the protection mechanism (parity or ECC) used for the primary copy indicates an error, the duplicate can provide the data as long as it is not corrupted. We implement two types of strategies based on this register duplication idea. In the "conservative strategy," we limit ourselves with the given register usage behavior, and duplicate register contents only on otherwise unused registers. Consequently, there is no impact on the original performance when there is no error, except for the protection mechanism used for the primary copy. Our experiments with two different versions of this strategy show that, with the more powerful conservative scheme, 78% of the accesses are to the physical registers with duplicates .The "aggressive strategy" sacrifices some performance to increase the number of register accesses with duplicates. It does so by marking the registers not used for a long time as "dead" and using them for duplicating actively used registers. The experiments with this strategy indicate that it takes the fraction of the reliable register accesses to 84%, and degrades the overall performance by only 0.21% on the average.

1. INTRODUCTION

Both consumer and business electronic product owners today are demanding high reliability in addition to high performance. With shrinking feature sizes, reducing supply and threshold voltages, transient errors are becoming an important problem. Such errors can occur even with zero-defect product and cause single-event upsets, which can eventually lead to application crash or wrong output. Moreover, the transient error problem is expected to be even more pressing in the future due to aggressive scaling-down of the supply voltages (V_{dd}), ever-scaling feature sizes, increasing clock rates, and the use of flip-chip packaging. Even if the error probability for a single transistor can be kept constant in the future, the overall error probability in a chip is going to increase in parallel with the number of transistors on a chip. While it is critical to put every effort to avoid these errors by careful circuit design and packaging, they can still occur and need to be addressed.

While recent research has mainly focused on the impact of errors on main memories [6], cache structures [5, 13, 14], and the datapath [24], the register file has largely been ignored. Although the register file itself does not occupy a large on-chip area, it is accessed very frequently. As a result, chances for a transient error in the register file to occur are high. In addition, a register error, if not taken care of, can quickly propagate to the other parts of the system. In light of this, several processor architectures employ error detection and recovery schemes in their register files, e.g. IBM G5 uses an ECC-based scheme [21]. In fact, soft errors in register files can lead to a large number of system failures [7, 20]. In addition, due to the high operation frequency, register files are not immune to other types of transient errors (e.g., errors due to inductive noise).

There are at least two obvious ways of protecting register files against transient errors. First, one can use parity to detect any odd number of bit errors. While a parity-based protection is not expensive to accommodate (from both performance and energy perspectives), it is limited since no error correction is provided. An ECC scheme, on the other hand, can correct single or multiple bit errors. However, this advantage comes at the expense of increased power consumption and latency. Even a simple ECC can take up to three times the delay of a simple ALU operation [27]. More importantly, the energy consumption of an ECC-based scheme can be as high as an order of magnitude larger than the energy consumed during a register access [17]. Realistically speaking, most of the errors that occur are single bit errors, and a scheme that can successfully correct single bit errors without incurring power/performance problems seems to be highly desirable.

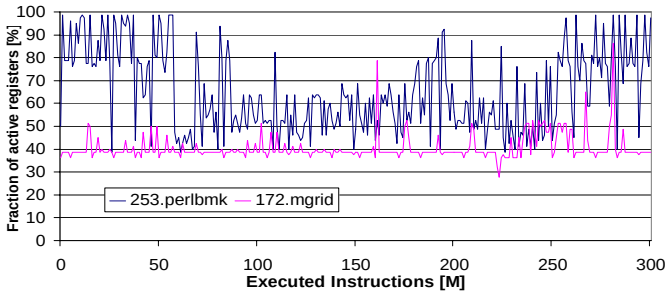

Figure 1. Fraction of active physical registers over the simulation time for the `253.perlbmk` and `172.mgrid` applications. See Table 1 for processor configuration.

Superscalar processors have been traditionally used to increase the performance in general-purpose architectures. Due to their complexity, their usage in embedded systems was limited. However, recently we are seeing increasingly more superscalar designs for embedded systems, e.g., STMicroelectronics [26], ARM [8], Renesans and Hitachi [9]. Our scheme targets such superscalar machines. In search for an error correction scheme that incurs small power and performance penalties, we studied the register usage of the Spec2000 benchmarks. Figure 1 plots the fraction of the physical registers that are *allocated* (i.e., that are mapped to a logical register or hold valid data) during the execution for `253.perlbmk` and `172.mgrid` benchmarks (the details of our experimental setup and simulation parameters will be given later). One can see from these plots that, while the register usage behavior changes during the course of execution, a significant fraction of the physical registers (in particular for the `172.mgrid`) are not used.

Based on this observation, this paper proposes to use unused physical registers as *duplicates* for the actively used ones. Specifically, we investigate two different strategies: *conservative* and *aggressive*. In the conservative strategy, we limit ourselves with the given register usage behavior of the application being executed. That is, we duplicate active registers in the unused ones without

* *This work is supported in part by the NSF Career Award #0093082.*

affecting the original execution cycle count of the application. We evaluate two different implementations of this conservative strategy. In the *conservative-base* (*CB*) version, whenever we allocate a new physical register, we create a duplicate for it, either in an unused register or over another duplicate. In the *conservative-enhanced* (*CE*) version, when creating a new duplicate over an existing one, we select the victim duplicate carefully, considering the frequency of register usage. Also, we initiate a copy operation whenever a register becomes available. A common characteristic of these two versions is that they do not increase the original number of execution cycles. As a consequence, when the register pressure is very high, they tend to perform no duplication. In comparison, the aggressive strategy creates a duplicate at the expense of performance. The main idea is to *predict* the cases where a physical register is not going to be used in the future, and (after writing its contents to memory) use it for duplicating the contents of an active physical register.

We used 20 applications (10 integer and 10 floating-point) from the Spec2000 benchmark to evaluate our strategies. Experimental results obtained using SimpleScalar [4] indicate that using the most powerful conservative scheme results in nearly 78% of the register accesses to have their duplicates (which is a good guarantee in case the protection mechanism of the primary register indicates an error). Our experiments also show that the aggressive scheme brings up the number of such reliable register accesses to 84% on the average, at the expense of a slight degradation in the original performance (0.21% on the average across all benchmarks).

The rest of this paper is organized as follows. The next section explains the default physical register allocation mechanism. Section 3 presents our experimental setup. Section 4 gives the details of the proposed duplication-based strategies, and presents experimental data showing their effectiveness. Section 5 discusses the related architectural-level work on reliability. Section 6 concludes the paper by summarizing our observations.

2. CONVENTIONAL REGISTER RENAMING

If the number of physical registers is sufficient, register renaming removes the write-after-read (WAR) and write-after-write (WAW) hazards, allowing higher levels of instruction-level parallelism (ILP) to be achieved at runtime. This is achieved by mapping logical registers to physical registers dynamically during the execution. Specifically, each new logical register value is mapped to a different physical register. Therefore, even if the instructions are executed out-of-order, no WAR or WAW hazards will occur.

For each fetched instruction, two tasks have to be performed to complete renaming: A new physical register has to be allocated for the destination register(s), and the source registers should be renamed to corresponding physical registers. There are several possible implementations to achieve this. In this work, we employ a scheme similar to that of Alpha 21264 [11], where the renamed and architectural register files are merged into a single register file, and there is a separate table for mapping logical registers to physical, each physical register can be in one of the following four states: not used or available (*NU*); used as an architectural register (*AR*); allocated, but not valid (*AN*): allocated-valid (*AV*).

Initially, only the physical registers that correspond to the logical registers are in the AR state, all the remaining registers are in the NU state. When a register is allocated for a logical register, it is transitioned to the AN state. Then, when the instruction in question writes the value to the register, the register is placed into the AV state. When the instruction completes, the register is placed into the AR state. If the instruction is cancelled (or squashed), the register is transitioned back to the NU state. Similarly, if the logical register is mapped to another register, the register is placed into the NU state after all the corresponding instructions complete. Register values can only be read if the physical register is in AV or AR states. There are usually three hardware structures to implement register renaming: The Mapping Table (MT) maps logical registers to physical registers; the Reservation Station (RS) keeps information about each fetched instruction; and the Register Renaming Circuit (RRC) stores the state of each physical register. The RRC can snoop the register file accesses to update the state values.

We modify this model to achieve a precise rollback and to keep track of "in-flight" instructions, i.e., the currently executing instructions that have a physical register mapped as a source register. If an instruction is squashed, we need to rollback only the corresponding registers to their original states and values. In recent Pentium family microprocessors [10] this is achieved with the help of a Reorder Buffer. We, on the other hand, extend the Reservation Station (RS) to hold the previous mapping information for the destination registers. We also keep track of whether the RS entry corresponds to an instruction that is the last consumer for a particular physical register and which RS entry was the previous last consumer for this source register. This instruction is used to transition the physical register state to AR or NU. Only the last consuming instruction changes the state of a register during its commit stage. If the last consumer instruction completes and the logical register is already mapped into another physical register, the register can be placed directly into the NU state; otherwise, it is transitioned to the AR state. If the last consuming instruction is squashed, we use the previous RS entry field to retrieve the information about the previous last consumer. If this instruction has already committed, we can change the state to AR or NU. In our experiments, we do not differentiate between AR, AN, and AV states: as far as our techniques are concerned, all these registers are considered as "*active*" registers. Hence, we only differentiate between registers that are not in use and registers that are active.

3. EXPERIMENTAL SETUP

In the rest of this paper, we explain each proposed technique followed by the presentation of its experimental results. Therefore, we first present our experimental setup.

Table 1. Simulated applications and important statistics: Number of execution cycles, total number of level 1 data cache accesses, number of register values read, number of register values written, the number of errors injected during the simulations, and the number of faults caused by these injected errors.

Appl.	cycle [M]	DL1 acc.[M]	Reg. reads[M]	Reg. writes[M]	Error injected[K]	Faults caused[K]
168.wupwise	260.1	93.4	550.82	284.55	32.9	30.9
171.swim	837.5	97.5	344.10	127.46	106.1	44.4
172.mgrid	492.9	109.8	285.96	48.28	62.4	57.8
173.applu	661.9	114.2	284.64	41.53	93.8	84.1
177.mesa	147.8	109.8	339.7	192.92	18.7	9.1
179.art	1845.7	102.8	309.8	125.65	233.7	92.4
183.equake	1407.6	127.2	436.50	183.93	178.2	139.9
188.ammp	762.8	116.2	501.86	195.35	96.6	40.6
189.lucas	567.2	72.0	338.17	154.46	71.8	55.9
301.apsi	308.6	111.8	571.27	230.48	39.1	33.3
FP. Average	729.2	105.5	396.28	158.46	92.3	58.9
164.gzip	200.8	71.8	480.1	309.7	25.4	17.4
175.vpr	682.3	118.8	428.2	248.9	86.4	61.2
176.gcc	376.0	126.7	459.7	270.5	47.6	19.6
181.mcf	2151.6	20.3	260.4	185.3	272.5	75.7
186.crafty	308.8	119.5	450.8	280.5	39.1	17.6
197.parser	576.8	89.2	498.1	289.8	73.0	38.2
253.perlbmk	261.5	108.3	419.3	240.4	33.1	14.5
254.gap	230.4	115.1	459.4	297.9	29.2	14.5
255.vortex	314.2	124.8	317.9	185.1	39.8	20.3
300.twolf	802.7	100.1	518.2	300.5	101.6	66.3
Int. Average	590.5	99.5	429.2	260.9	74.8	34.5
Average	659.9	102.5	329.9	178.0	83.6	46.7

3.1 SIMULATION PARAMETERS AND BENCHMARKS

The SimpleScalar/Alpha [4] simulator is used to evaluate the proposed techniques. The necessary modifications to the simulator

have been implemented to perform register renaming, error injection, and the proposed error protection strategies. Some of the schemes use the *selective replay capabilities* that exist in some modern microprocessors such as Intel Pentium 4 [12]. Selective replay is used to selectively rollback instructions that were not able to execute correctly because of a misprediction. Therefore, we have also made changes to SimpleScalar to simulate a realistically sized issue queue, and to simulate a realistic scheduler under selective replay. The base architecture is a 4-way superscalar processor with 32 logical and 80 physical integer and floating point registers, 16 KB, 4-way associative level 1 instruction and data caches and a 256 KB, 8-way unified level 2 cache.

We simulate 10 floating-point and 10 integer benchmarks from the SPEC2000 benchmarking suite. We simulate 300 Million instructions after fast-forwarding an application-specific number of instructions as proposed by Sherwood et al. [23]. The important statistics for our applications are given in Table 1. We see from this table that the number of register accesses is much higher than the number of L1 data cache accesses, which indicates that the chances for a transient error in the register file to be consumed are higher than that of a cache error. The last two columns of the table give the number of errors we inject into the register file during the simulations and how many of those injected errors cause a fault.

3.2 Error Injection

To test the effectiveness of the error protection techniques proposed in this paper, we inject errors into the register file. These errors can change the value stored in the register file (representing soft errors), or they can occur during the reading or writing the register value (representing other types of transient errors). The former is achieved by using a random number generator to trigger an error in a random location in the register file. This generator is invoked at every simulation cycle. The error probability is set to 10^{-7} per each bit in the register file in accordance with literature [15]. All the base experiments use this probability. Later, to conduct a sensitivity analysis, we have also performed some experiments with a larger probability. Similarly, for the latter type of errors, in every access to the register file (including register writes), we generate an error with the same probability. In accordance with other studies, the probability of two-bit errors is set to 10^{-9} and the probability of three-bit errors is set to 10^{-11} [15].

4. Error Protection Strategies

4.1 Conservative Strategies

The conservative strategies proposed in this work utilize only the physical registers that are not used by the register renaming mechanism. Consequently, they do not degrade the performance compared to the base architecture.

4.1.1 Conservative-Base (CB)

In the base architecture, the RRC has information about the physical registers that are not used. Therefore, when we allocate a new physical register, we also allocate a register that will be used for copying the register value[1]. The copy register name is placed into the RUU (or the Reservation Station). At the writeback pipeline stage, the copy register is also written. In CB, this is the only time when a copy can be generated. Note that, the RRC can later allocate the copy register to be used for storing copies of other register values. In addition, the copy register can also be activated, i.e., logical registers can be mapped to it. Therefore, even if during the execution of the program all the physical registers are active, there is no performance penalty caused by CB.

Figure 2 depicts the structures that are modified for CB. There are three important modifications. First, the RRC is enhanced to select a copy register and store the copy register name for any physical register that has a copy. Also, a physical register can be in an additional state called *copy*, which indicates that it is used as a copy register (duplicate). For CB, this information is only needed during an error. Second, the Reservation Station is also enhanced to store the name of this copy register to enforce the copy operation during the execution of the instruction. Therefore, the path between the register file and the RS should be modified to contain this information. Finally, we need to make a modification to the register file as well. Specifically, it should be enhanced to perform the copy operation. For CB, copy operations are performed only during the write operations. Therefore, we can add a "copy" port for each write port in the register file. Any input in the copy port would force the corresponding write value to be copied in the given register name. Note that, additional ports can potentially have an impact on the access time of the register file. *However, the complexity of implementing the copy operation discussed here is expected to be less compared to having an additional write port, because the values written to the registers will be the same. Hence, we only need to receive two register numbers and one value as input and write the value to both the registers.* In addition, compared to alternative reliability measures (such as ECC[3]), the increased complexity of this addition would be smaller. In any case, if this extra delay is intolerable, one option might be to perform the copy operation at the background (e.g., one cycle after the actual write).

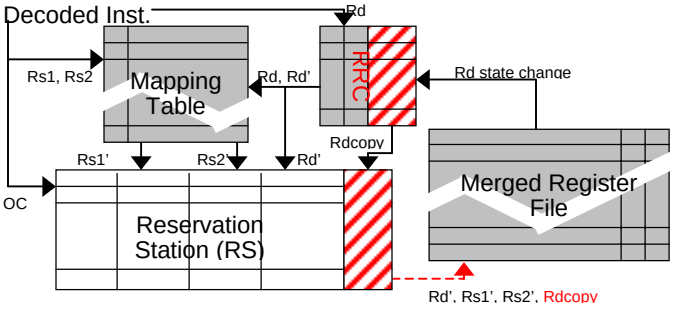

Figure 2. Register renaming for CB. The straddled area indicates the additional hardware required. The addition to RS is used to store the name of the copy register for the destination register. The RRC stores information about the copies as well.

Irrespective of the error detection mechanisms, we can use two metrics that can demonstrate how our duplication-based schemes are performing in practice. The first of these is called the *duplication success rate*, which is defined as the fraction of successful duplication attempts. The second metric is the *reliable read rate*, which gives us the fraction of time we read a register for which a duplicate exists. Of these, the reliable read rate is more important since we ultimately want to maximize the number of times where the accessed physical register has its duplicate. Figure 3 presents the results for both statistics. Figure 3(a) gives the duplication success rate for CB (due to lack of space, we only present the results for two applications and averages). We find that the duplication success rate is around 85% for the integer benchmarks and 88% for the floating-point benchmarks. These values are reasonable since the only time a duplication attempt can fail is when all the physical registers are active, which happens rarely in a typical execution. We also see that majority of the duplicates are made to empty physical registers. This is mainly because in this scheme when a duplicate is overwritten (by another duplicate or a primary copy), it is not restored. When we look at the reliable read rates given in Figure 3(b) (marked as *has copy*), we find that on the average 71% of the reads are made to the registers with duplicates. This result tells us that 71% of the time we can correct odd number of bit errors if the duplicate is clean (assuming that the registers are protected using parity).

CB has two main drawbacks. First, in many cases, it might happen that a register can lose its duplicate but continue to be read many times in subsequent computation. Since in CB the duplicates are

[1] The register that stores the copy value is called the *copy register*.

created only at writes, all such accesses will be counted as "unreliable reads." Since CB allocates copy registers on write only, during the periods with high register pressure, most copies are overwritten and they are reallocated a copy register when the pressure reduces. The second drawback is that, when it is creating a new duplicate in a register that holds another one, the selection of this victim register is done in some predetermined order.

4.1.2 *CONSERVATIVE-ENHANCED (CE)*

This scheme tries to eliminate the drawbacks of the CB by initiating copy operations whenever a register becomes available. In addition, CE tries to alleviate the second shortcoming of CB by being more careful in selecting the copy registers. More specifically, for each physical register, CE keeps track of the last time the register has been accessed. Then, if a copy has to be overwritten, the copy belonging to the "oldest" physical register is selected. The oldest in this context is the physical register that has not been accessed for the longest duration at the time of the decision. CE also uses a similar approach to decide on the physical register to be copied in the case of copy generation due to copy-on-free (i.e. the copy operation performed when a register becomes available).

To capture the age of a physical register, we use a 7-bit global counter that is incremented at every 1000 cycles. When a register is accessed, the value of the global counter is stored in the corresponding register access time field in the RRC. When a decision is made, this value is used to select the copy register to be overwritten or the physical register to be copied. If the counter rolls over (which will happen at every 128,000 cycles), we check the most significant bits of the counter values of the physical registers. The physical registers with the MSB value being zero (i.e., the registers that have not been accessed for more than 64,000 cycles) are marked using an additional "too-old" bit in the RRC. When a copy register needs to be overwritten and there is a physical register with a copy that has the too-old bit set, its copy is overwritten.

The duplication success rate and reliable read rate for this scheme are given in Figures 4(a) and 4(b), respectively. One can observe from these results that CE in general generates better results than CB. Specifically, CE achieves a 78% reliable read rate, and 65% of the copies overwrite other copies. Intuitively, a register that has not been accessed for more than 10,000 cycles has less chance of being accessed again soon compared to a register that has been accessed in last 10 cycles. In Section 4.2, we study these probabilities, and present empirical data that this intuition is indeed correct. Therefore, by having copies for registers that are recently accessed, CE increases the reliable read rates.

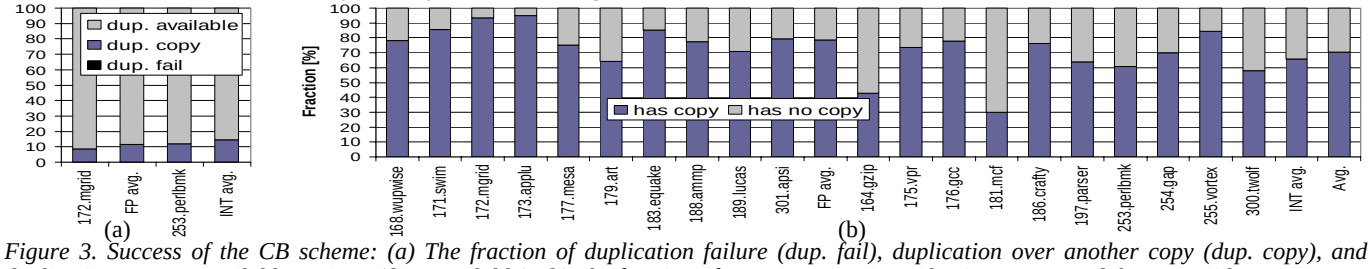

Figure 3. Success of the CB scheme: (a) The fraction of duplication failure (dup. fail), duplication over another copy (dup. copy), and duplication over an available register (dup. available), (b) The fraction of accesses to registers having a copy and the ones without a copy.

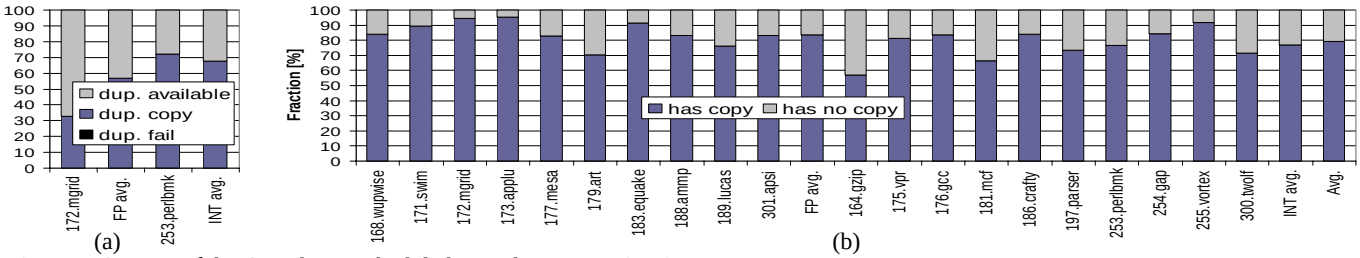

Figure 4. Success of the CE scheme. The labels are the same as in Figure 3.

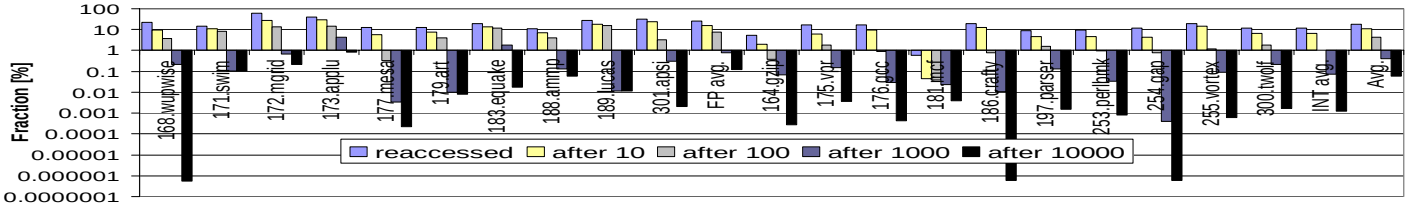

Figure 5. Register re-usage: Fraction of registers accessed after being put into the AR state (reaccessed), fraction of registers accessed after 10 cycles (after 10), after 100 cycle (100), after 1000 cycles (1000), and after 10000 cycles (after 10000) of going into the AR state.

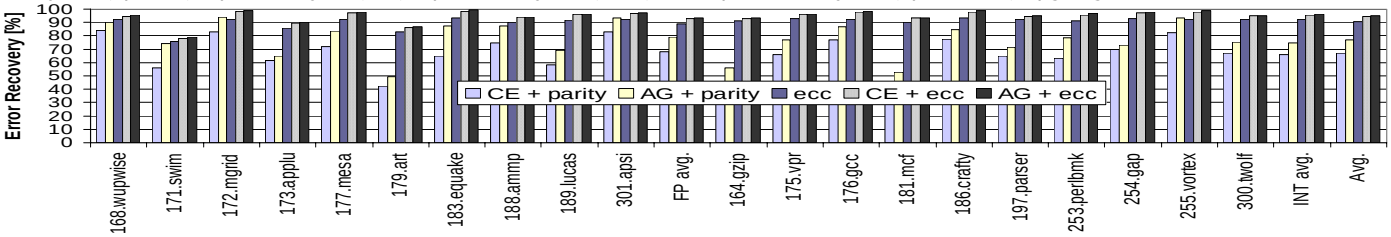

Figure 6. Error recovery rates for Aggressive scheme. AG + parity is the aggressive scheme with pure-parity based error detection, AG + ecc is the aggressive technique along with the ECC error recovery/detection.

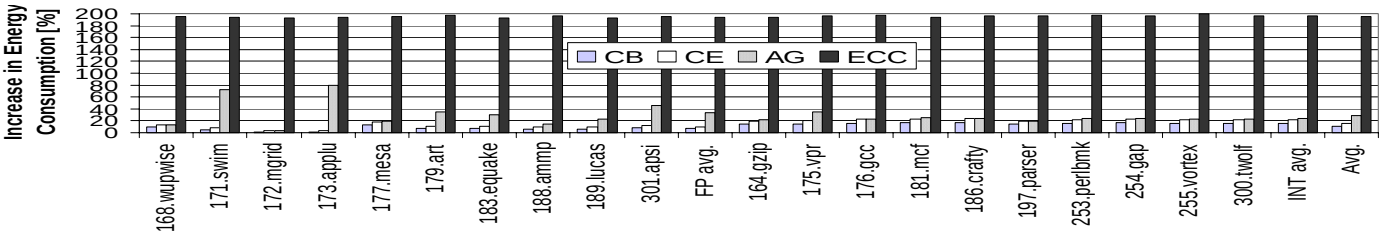

Figure 7. Increase in energy consumption for different schemes. The energy numbers are relative to the consumption in the register file.

4.2 AGGRESSIVE STRATEGY (AG)

The partial success of CE is a strong motivation to look for more aggressive schemes. Particularly, an aggressive scheme can address the failures incurred by the lack of a duplicate. We started to search for such a strategy by collecting further statistics on register usage. Figure 5 shows the fraction of active physical registers accessed after going into the AR state. Note that small values are highlighted in the lower portion of the graph. We see from these results that, on the average, only 18% of the registers are re-accessed after going into the AR state. This ratio drops to 10.8%, 4.4%, 0.4%, and 0.06% after 10, 100, 1,000, and 10,000 cycles after going into the AR state, respectively. In other words, if we wait long enough, it is likely that contents of a given physical register are dead, and thus can be used for holding the duplicate of an active register.

Architecturally, AG is not significantly different from CE. The RRC is enhanced to implement the eviction decision. This will be explained further in the following discussion. In addition, the physical register states are also changed. Instead of five states, the AG scheme uses two additional states: *victim-copy* and *victim-available*. Victim-copy indicates that the RRC has evicted this physical register and uses it as a copy register. Victim-available, on the other hand, indicates that the physical register is evicted, but it is not used at the moment. Victim registers are never mapped to logical registers. If we had mapped it to other logical registers, we would have needed a complicated rollback mechanism to change the mapping. Once a register is selected as a victim register, the RRC introduces a dummy store operation into the pipeline. We assume that predetermined portions of the address space are allocated for copying the registers. The RRC then checks for source registers addressed to this physical register. If such an instruction is fetched, the RRC introduces a dummy load operation in front of this instruction to read the value back to the register file. Since the original register can only be used as a copy register, this load is guaranteed to not change any logical register mappings. If, instead, the logical register is overwritten, the register will be placed into the NU (if it is victim-available) or Copy (if it is victim-copy) states.

AG is conservative in the register evictions. If the number of available registers goes above 10% of the available physical registers, the evicted registers are read back to the register file. It should be emphasized that even a misprediction in our context does not cause a correctness problem, as we always write the contents of the register to the memory before it is modified. However, a high misprediction rate can cause an increase in the original execution cycles. Therefore, the threshold value after which the contents of a register are pronounced dead should be kept large enough. In our experiments, we mark the registers that are not accessed for more than 10,000 cycles as dead. This results in increased reliability with minimal performance impact. Due to limited space, we do not present the results for the duplication success rate and reliable read rate for the AG. In summary, the reliable read rate for the aggressive scheme is 84% on the average. The duplication success rates are reduced compared to CE. This is an expected result as, with the evicted registers, the RRC has more registers that can be used for copying, thereby reducing the number of copy overwrites.

4.2.1 ERROR RESILIENCE RESULTS

Note that, all the results presented so far were for simulations without injecting any errors. While these results clearly showed that our strategies are successful in providing reliable reads, it is also important to evaluate their behavior under errors. To do this, we inject transient errors into the register file using the method explained in Section 3.1. The graph in Figure 6 presents the error resilience results for AG when used with parity (AG + parity) and with ECC (AG + ecc). We also present the results for CE + parity, CE + ecc, and pure ECC schemes. Note that, for the aggressive scheme, we assumed that the memory is reliable, i.e., if the register does not contain any errors while it is written to the memory, it will still have no errors when it is read back. The results indicate that the aggressive scheme performs better, in terms of error recovery, than CE. Particularly, AG + parity and AG + ecc can recover over 77.1% and 95.0% of the transient errors, respectively. We have also performed an analysis of the failed recoveries for AG + parity. We do not present the detailed results here; but, they revealed that most failures (37.1%) are due to lack of duplicates, and the multi-bit errors and the corrupted copies constitute 30.5% and 32.4% of the failures, respectively.

4.2.2 PERFORMANCE RESULTS

Unlike the conservative strategies, AG has a performance penalty that should be quantified as well for a fair comparison. Therefore, we measure the *misprediction rate* for AG. We define the misprediction rate as the number of times we pronounce a physical register as dead but the register is later accessed. The misprediction rates for integer and floating-point benchmarks are 0.22% and 0.19%, respectively, indicating that our prediction mechanism works very well with a threshold value of 10,000 cycles. To see the impact of this misprediction rate on the overall performance of our applications, we recorded execution cycles. The average performance degradation is around 0.09% across all applications.

4.3 ENERGY CONSUMPTION RESULTS

After having discussed the reliability and performance results in detail, we now focus on energy consumption of different schemes. Figure 7 gives the register file energy consumption for four schemes (CB in conjunction with parity, CE in conjunction with parity, AG in conjunction with parity, and the pure ECC scheme), as normalized values with respect to the pure parity-based scheme for an error-free execution (the energy results with errors injected are very similar since the number of errors is very small compared to the total number of register file accesses). To measure the energy consumed by the ECC scheme, we simulated the error detection/correction events. Then, we used published energy consumption values [17] to find the energy impact of the techniques. On average, an ECC operation consumes 206% of the energy consumed for a register file access. Also, it should be emphasized that the energy consumption values given for CB and CE include all the energy overheads associated with these schemes. Specifically, we consider the register write operations due to copies, the energy overhead of the register file due to the additional copy port, increased size of the reservation table and the RRC for all the techniques; the global-counter and 8-bit counter stores for CE; the extra memory accesses, effects of selective replay, and increased number of instructions executed for the aggressive strategy. We see from these results that the conservative strategies are much more energy efficient than the pure ECC scheme. This is because the ECC scheme pays an energy price (over the pure parity-based protection) at each register access (i.e., whether we have error or not). On average, CB, CE, AG, and ECC increase the energy consumption by 11.1%, 15.7%, 28.8%, and 195.6% respectively. It should also be

emphasized that, while a 15.7% increase in the register file energy consumption (for CE) may seem significant at first glance, the register file energy itself typically constitutes a small fraction of the overall system energy. Therefore, one may expect the impact on the overall system energy consumption to be within tolerable limits. Overall, these results clearly indicate that, while a pure ECC-based protection mechanism might be a good candidate from the reliability angle, it is certainly not a good candidate when one considers power consumption. Note that, even if the ECC computation is done at the background (i.e., with no performance penalty) as has been assumed here, it is still not possible to hide its energy overhead.

4.4 DISCUSSION

In this paper, we have presented a scheme for increasing the robustness of register files to transient errors. The main idea was to duplicate the values of the activate registers in the registers that are not used. One could also come up with alternate techniques for duplicating the register values. One such scheme would be to replicate the register file and have a second "copy" register file where the writes to the main register file are snooped and performed on the copy register file. The main drawback with this approach would be doubling the register file size. Although the register file itself constitutes a small fraction of the overall chip area, it resides in a critical segment of the datapath, and increasing the area in this segment can have a negative impact on the overall performance of the processor. Note that, in comparison, our approach tunes the aggressiveness of protection based on the dynamic demands on the physical registers.

5. RELATED WORK

Designing for transient-error tolerance has traditionally been considered in the context of systems that operate in high-radiation environments or in outer space, where there is a heavy concentration of alpha-particles and atmospheric neutrons [28]. Research from IBM showed that computer systems are susceptible to transient faults induced by these particles [25]. More recently, designing computer systems for resiliency [16] to transient faults has gained greater significance due to the combined effect of higher integration densities, lower voltages, and faster clock frequencies. Redundancy is a frequently used technique for providing fault tolerance. Spatial redundancy may involve a complete duplication of all hardware components as in the HP NonStop Himalaya machine [1]. The IBM POWER4 [3] provides an extensive hierarchy of checkers distributed in all the major sections of the processor. Temporal redundancy has been proposed for both superscalar and SMT paradigms [18, 19, 22]. The DIVA architecture takes a slightly different approach wherein a special checker-processor is used at the commit stage of the main core's pipeline to verify the correctness of the instructions being committed [2].

6. CONCLUDING REMARKS

This paper proposes and evaluates different strategies for increasing the resilience of register files to transient errors. These strategies are based on the observation that at a given time a significant fraction of the physical registers do not hold valid data, and can thus be used as placeholders for duplicates of the actively used registers. Our conservative schemes do not impact original performance of applications, and duplicate registers using only otherwise unused registers. We found that the most effective conservative scheme (CE) has around a 78% reliable read rate, and recovers from the 67% of the cases where a pure parity-based scheme fails. Our experiments with the aggressive strategy showed that it takes the error recovery rate to 77%, at the expense of slight (0.21%) degradation in the original performance. It is to be emphasized that errors occur rarely. Therefore, one of the most important issues is not to incur too much additional performance/energy overhead in the normal (error-free) case. Our conservative and aggressive strategies achieve this. Hence, we believe that the duplication-based recovery schemes are attractive solutions under these requirements.

REFERENCES

[1] Albonesi, D. H. Selective cache ways. In *Int. Symposium of Microarchitecture*, Nov. 1999. Haifa / Israel.

[2] Austin, T. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Intl. Symp. on Microarchitecture*, Nov. 1999.

[3] Bossen, D. C., J. M. Tendler, and K. Reick., *POWER4 System Design for High Reliability*. IEEE Micro, March 2002, **22**(2): p. 16-24.

[4] Burger, D. and T. Austin, *SimpleScalar Tool Set, Version 2.0*. June 1997, University of Wisconsin.

[5] Chen, C. and A. K. Somani, *Fault Containment in Cache Memories for TMR Redundant Processor Systems.* IEEE Transactions on Computers, March 1999, **48**(4): p. 609-623.

[6] Dell, T. J., *A White Paper on the Benefits of Chipkill-Correct ECC for PC Serve Main Memory*. Nov. 1997, IBM Microelectronics Division.

[7] Dupont, E. Soft Error: A new skill in semiconductor business. In *International Reliability Physics Symposium*, 2003.

[8] EETimes. Two call on superscalar CPUs for handset apps, Oct. 2003, http://www.eetimes.com/futureofsemis/showArticle.jhtml?articleId=16502123&kc=6255.

[9] Global-Electronics. Renesas and Hitachi announce development of 32-Bit RISC CPU Core for Embedded Systems, http://www.global-electronics.net/id/23262/CMEntries_ID/46656.

[10] Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, *The microarchitecture of the Pentium 4 processor*. 2001.

[11] Kessler, R., *The Alpha 21264 Microprocessor*. IEEE Micro, Mar/Apr 1999, **19(2)**.

[12] Kim, I. and M. H. Lipasti. Understanding Scheduling Replay Schemes. In *International Symp. on High Performance Computer Architecture*, Feb. 2004.

[13] Kim, S. and A. K. Somani, *An Adaptive Write Error Detection Technique in On-Chip Caches of Multi-Level Caching Systems.* Journal of Microprocessors and Microsystems, March 1999, **22**(9): p. 561-570.

[14] Kim, S. and A. K. Somani. Area Efficient Architectures for Information Integrity in Cache Memories. In *Intl. Symp. on Computer Architecture*, 1999.

[15] Li, L., V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: a data cache perspective. In *International Symposium on Low Power Electronics and Design*, 2004.

[16] Patel, S. J., Z. Kalbarczyk, R. K. Iyer, W. Magda, and N. Nakka. A Processor-Level Framework for High-Performance and High-Dependability. In *Workshop on Evaluating and Architecting Systems for Dependability*, 2001.

[17] Phelan, R., *Addressing Soft Errors in ARM Core-based SoC*. Dec. 2003, ARM Ltd.

[18] Rashid, F., K. K. Saluja, and P. Ramanathan. Fault tolerance through re-execution in multiscalar architecture. In *International Conference on Dependable Systems and Networks (DSN)*, June 2000.

[19] Ray, J., J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *International Symposium on Microarchitecture (MICRO)*, Dec. 2001.

[20] Rebaudengo, M., M. S. Reorda, and M.Violante. An Accurate Analysis of the Effects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor. In *Design, Automation and Test in Europe (DATE)*, 2003.

[21] Reinhardt, S. K. and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Intl. Symp. on Computer Architecture*, 2000.

[22] Sato, T. and I. Arita. Tolerating Transient Faults through an Instruction Reissue Mechanism. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*, Aug. 2001.

[23] Sherwood, T., E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Intl. Conf.on Parallel Architectures and Compilation Techniques*, Sep. 2001.

[24] Shivakumar, P., M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Intl. Conf. on Dependable Systems and Networks (DSN)*. Bethesda, MD.

[25] Srinivasan, G. R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview.* IBM Journal of Research and Development, Jan. 1996, **40**(1): p. 77-89.

[26] STMicroelectronics. STPC Microcontrollers - Overview, 2004, http://www.st.com/stonline/products/support/micro/stpc/home.htm.

[27] Tremblay, M. and Y. Tamir. Support for Fault Tolerance in VLSI Processors. In *Intl. Symp. on Circuits and Systems*, 1989. Portland, Oregon.

[28] Turmon, M., R. Granat, and D. Katz. Software-implemented fault detection for high-performance space applications. In *International Conference on Dependable Systems and Networks (DSN)*, June 2000.