Engineering Over-Clocking: Reliability-Performance Trade-Offs for High-Performance Register Files

Gokhan Memik[1], Masud H. Chowdhury[2], Arindam Mallik[1], Yehea I. Ismail[1]

[1] Dept of Electrical and Computer Engineering
Northwestern University
{memik, arindam, ismail}@ece.northwestern.edu

[2] College of Engineering,
University of Illinois at Chicago
masud@ece.uic.edu

ABSTRACT

Register files are in the critical path of most high-performance processors and their latency is one of the most important factors that limit their size. Our goal is to develop error correction mechanisms at the architecture level. Utilizing this increased robustness, the clock frequencies of the circuits are pushed beyond the point of allowing full voltage swing. This increases the errors observed due to noise and other external factors. The resulting errors are then corrected through the error correction mechanisms. We first develop a realistic model for error probability in register files for a given clock frequency. Then, we present the overall architecture, which allows the error detection computation to be overlapped with other computation in the pipeline. We develop novel techniques that utilize the fact that at a given instance many physical registers are not used in superscalar processors. These underutilized registers are used to store the values of active registers. Our simulation results show that for a fixed architecture the access times to the registers can be reduced by as much as 80% while increasing the number of execution cycles by 0.12%. On the other hand, by reducing the register file access pipeline stages by 75%, the average number of execution cycles of SPEC applications can be reduced by 11.5%.

Keywords: Reliability, Fault-Tolerant Computing, Adaptive Systems.

1. INTRODUCTION

Over the last decade, in spite of the complexities of new manufacturing technologies and increasingly complicated architectures, designers have been able to steadily increase the performance of high-end microprocessors. This improvement is achieved through optimizations at the architecture level (such as aggressive pipelining strategies) and at the circuit level (such as smaller feature sizes). As we move into deeper sub-micron technologies, the complexity of pushing the circuit performance further becomes an important obstacle. To achieve better performance, there is an increasing need for collaboration of higher level (e.g. microarchitecture-level) and circuit-level optimizations. In this work, we present such a collaborative optimization. Particularly, we provide architectural structures to increase the robustness of the register files in high-end processors, thereby allowing the designers to push the operating frequencies further[1]. The reduced delay times usually result in an increase in the number of errors observed due to noise and other external effects. However, the architectural structures proposed allow the processor to recover from these errors efficiently. Our goal in this paper is to investigate this trade off between the register file access delay and its reliability and allow architects to find the optimal operation frequency. Specifically, in this paper we make the following contributions:

- We present a realistic model that determines the probability of an error for a given cycle time of a register,
- We present simulation results showing that a significant fraction of the registers are not utilized for a representative processor architecture,
- We propose a novel error recovery scheme that exploits these underutilized registers,
- We study how different error recovery mechanisms can be employed by a high-performance microprocessor,
- We present simulation results investigating an optimal point for trading off the reliability for reducing cycle time of a register file in a representative architecture.

High-performance processors are aggressive: they try to fetch and execute multiple instructions per cycle, are speculative. In such processors, there are two important hardware loops that affect performance: Branch Loop and Memory Loop [1]. The Branch Loop includes the stages between when a prediction for the outcome of a branch instruction is made and when the outcome of the branch instruction is found. The Memory Loop includes stages between a load operation is scheduled and the cache access is made. The lengths of these loops are arguably the most essential components in the overall performance of a processor [7]. The longer the loop, the longer it will take to recognize a misprediction and recover from it. For all high-performance microprocessors, register file access stage(s) are in both of these loops. Hence, the access latency to a

[1] Note that, we do not vary the supply voltage (V_{dd}). We change the input clock frequency.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

register file is likely to have a significant impact on the overall performance. Our proposed schemes aim to achieve reduced access latency for the register file. Particularly, in Section 5, we show that the number of register file access pipeline stages can be reduced by as much as 75%, thereby reducing the average number of execution cycles of SPEC applications by 11.5% on average. By allowing the register file to operate at higher frequencies, we will allow larger register files to be implemented.

In the next section, we present a study investigating the relation between the cycle time and error probability in register file. Section 3 gives an overview of how the errors are detected and corrected. In Section 4, we discuss our novel error correction schemes. Section 5 presents the experimental results. In Section 6, we overview the related work and Section 7 concludes the paper with a summary.

2. FREQUENCY VS. RELIABILITY

We present an analytical framework, which relates reliability with overclocking scheme used in the register file. This section discusses the model that we have used in our work.

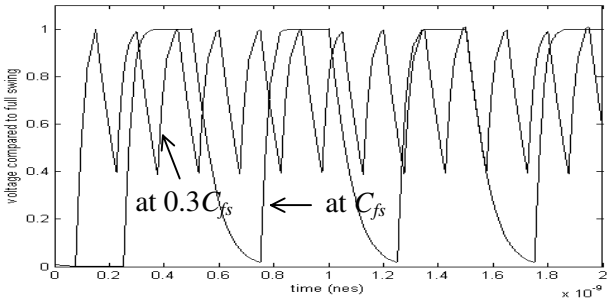

Figure 1. Voltage at a circuit node at two different frequencies

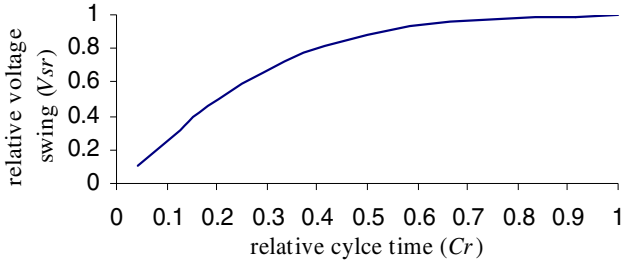

Figure 2. Decrease of voltage swing with increase of frequency

Injection of noise into a circuit node causes a signal deviation at that node. This signal deviation will affect the operation of the circuit or circuit block driven by the victim net. A functional failure is possible when induced noise is propagated and wrongly evaluated at the primary output. The parameters that determine if there will be a logic error are (i) the amplitude and the duration of the noise pulse, (ii) the type of the victim node and the circuit connected to the victim node, and (iii) the signal condition on the affected node. It is important to note that with increasing clock frequencies, a circuit node may suffer from reduced voltage swing, since there is not enough time to fully charge or discharge the load capacitance. C_{fs} in Figure 1 is the clock cycle time required to obtain the full voltage swing (V_{fs}) from zero to V_{dd}. Note that the supply voltage is kept constant at V_{dd}.

Figure 10 illustrates the decrease of voltage swing (Vs) with the decrease of clock cycle time (C). The clock cycle time and the voltage swing are normalized against the clock cycle at full swing (C_{fs}) and the full swing voltage (V_{fs}), respectively. The relative voltage swing is defined as $V_{sr} = V_s/V_{fs}$ and the relative cycle time $C_r = C/C_{fs}$. If the voltage swing changes, all the signals become faster by the same ratio independent of the capacitive load at a circuit node. Note that the change of voltage swing slows down at longer clock cycle time. This shape correctly maps the change of actual signals on-chip with time. Any signal at a circuit node rises quickly at the beginning and as the signal reaches close to the full swing value it takes longer time for a certain change. The curve in Figure 1 has been produced by simulating a chain of gates driven by an inverter at different frequencies with constant supply voltage V_{dd}.

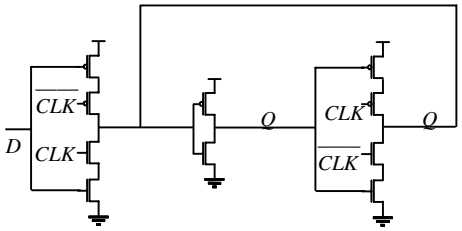

Figure 3. A simple D Flip-Flop

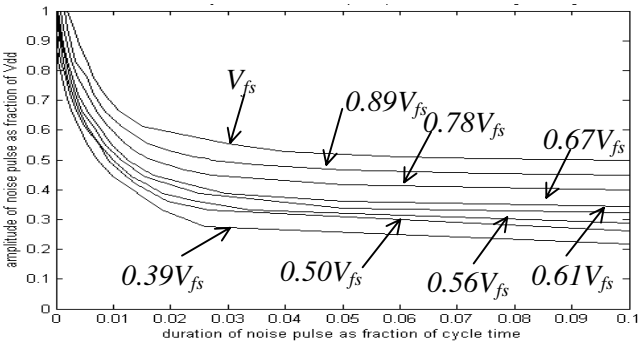

Figure 4. Noise immunity curves of a D flip-flop at various voltage swings

With a reduced signal level, a circuit node is more likely to suffer from logic failure due to a certain level of noise. Therefore, increasing frequency leads to higher probability of logic failure at a circuit node due to reduced voltage swing. The main advantage of static logic over dynamic logic is its robustness under the influence of noise. But static logic may suffer from logic failure if there is a feedback loop. A static D flip-flop (as in Figure 3), which

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

is common in registers, has a feedback loop that cannot recover from noise-induced errors. In these types of circuits there are three possible points where noise can be injected: the input, the clock and the feedback loop. The feedback loop is the most sensitive to noise. Even a small noise pulse on the feedback loop when the clock is falling or inactive will be propagated repeatedly through the loop and may ultimately destroy the logic information stored in the flip-flop. A set of noise immunity curves for the D flip-flop in Figure 3 is presented in Figure 4, which plots the relative noise duration (D_r) against the relative noise amplitude (A_r) at various voltage swings. Noise pulses of various amplitudes and durations have been injected into the feedback loop of a D flip-flop at different voltage swings, while keeping V_{dd} constant. SPICE simulations were used to determine the set of noise amplitudes and durations that cause a logic failure for different voltage swing levels. The area above each curve in Figure 4 represents the amplitudes and durations of a noise pulse that can cause logic failure. Hence, the lower the voltage swing the larger the area of noise amplitudes and durations that can cause an error. The relative noise amplitude is defined as $A_r = A/V_{fs}$, where A is the amplitude of the noise pulse, and the relative duration of noise $D_r = D/C_{fs}$, where D is the duration of the noise pulse. The highest curve is for the full voltage swing V_{fs} (swing from zero to V_{dd}). The lower curves illustrate noise immunity at voltage swings smaller than the full swing. It is important to note that the noise amplitudes and durations are not equally probable. The probability of smaller noise amplitudes and noise durations are higher than larger amplitude pulses with longer duration.

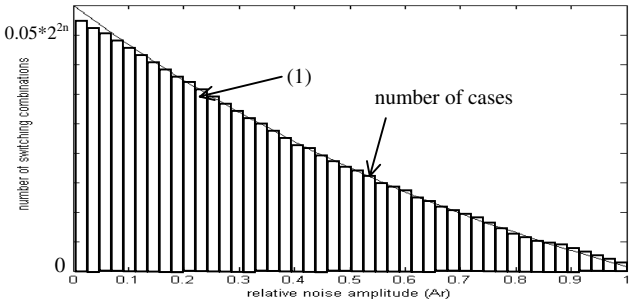

Figure 5. Noise amplitude at various switching combination of neighboring lines of a victim line

Consider a victim line, which has n neighbors significantly coupling to it. For noise injection into the victim line the total number of switching combinations of the neighboring lines is 2^{2n}. Only one switching combination results in the worst-case noise amplitude, which occurs when all the neighboring lines switch in the same direction. However, the number of cases where the effects of most of the neighboring lines cancel each other resulting in small amplitude of noise is large. We have found the number of switching cases between these two limiting cases, which result in a certain noise amplitude range. The results are plotted in Figure 5. This distribution can be approximated by an exponential as in (1).

$$\text{Number of cases} = K_1 e^{-K_2 A} \tag{1}$$

The exact constants K_1 and K_2 depend on the number of lines (n) coupling to the victim line. For large n (greater than 16) this curve saturates to continuous probability distribution of the form

$$P(A_r) = 28.8 * e^{-28.8 A_r} \quad \text{where} \quad 0 < A_r < \infty \tag{2}$$

$$P(D_r) = 10 \quad \text{for} \quad 0 < D_r < 0.1$$
$$P(D_r) = 0 \quad \text{for} \quad 0.1 \le D_r \tag{3}$$

The probability distribution of noise duration can be given by (3). The reason why D_r is uniformly distributed between 0 and 0.1 is that this is the range of rise time on chip as a ratio of the cycle time. Note that the noise duration is limited by these rise times, since noise occurs due to capacitive and/or inductive coupling of switching line to a victim line.

Once an aggressor signal settles, the noise pulse ends. Using equation (2) and (3), the probabilities (PE) of logic failure for a D flip-flop at different voltage swings have been obtained by the integration of the probabilities of noise pulse above each curve of Figure 6. Figure 6 plots the probabilities of logic failure against the relative voltage swings (V_{rs}). The probability number at full voltage swing are consistent with industrial and test data [23].

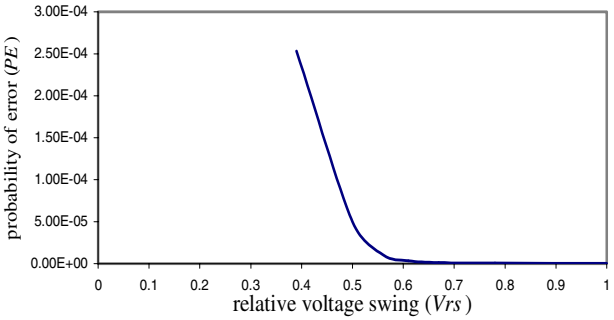

Figure 6. Probability of error at different cycle time

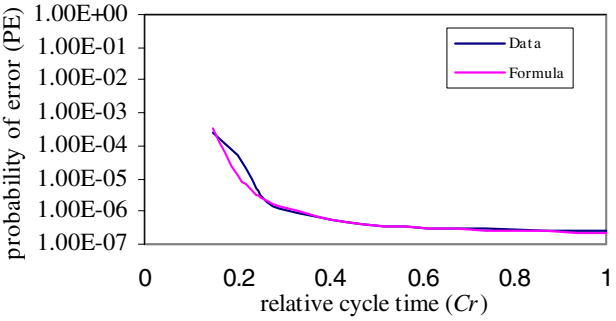

Figure 7. Probability of error at various voltage swings

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

The probability of error versus cycle time in Figure 7 has been obtained by the voltage swing variable from the two relations: cycle time versus voltage swing (Figure 2) and probability of error versus voltage swing (Figure 6). The relative cycle time Cr is always less than 1 for lower voltage swings. Similarly we can define relative frequency $F_r = f/f_{fs} = 1/C_r$, where f is the frequency and f_{fs} is the frequency at full voltage swing. PE is a single bit probability of error and is a function of how fast a circuit is driven by allowing the voltage swing to decrease. The formula below shows the relation between PE and Cr and Fr.

$$P_E = 2*10^{-7} * e^{\frac{1}{6*C_r^2}} = 2*10^{-7} * e^{\frac{F_r^2}{6}} \qquad (4)$$

These formulae have been found by curve fitting for the data of the above curves. The curves in Figure 7, showing the data and the curve fitted formula, illustrate the accuracy of the formula. Note that if the circuit is pushed enough not to allow any voltage swing, the error probability will be 1. However, the circuit is never pushed to these limits. Note that, this particular fault model is applicable for a specific circuit element, register file in current work. The other parts of the circuit won't follow the same fault model. However, using similar procedure, it is possible to come up with accurate fault models for other parts of the processor. In our earlier studies we have developed a fault model which predicts the fault occurrence probability in the data cache [11].

The overclocking of the register file can be implemented either statically or dynamically. For static implementation, the clock rate would be decided at the design time. This will be performed by setting the clock period higher than the estimated delay. This scheme won't require a separate clock for the register file. Dynamic implementation, on the other hand, would adjust the clock of the system to a higher (lower) value as the amount of error is below (above) a predetermined threshold value. However, this dynamic adjustment has a high hardware overhead. Hence, in our work we utilize a static overclocking scheme.

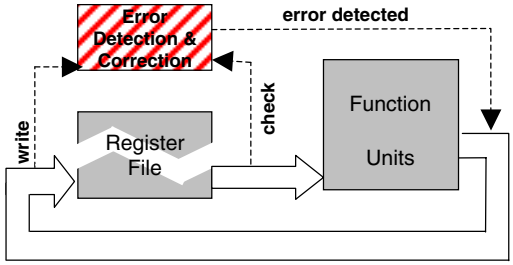

Figure 8. System overview of error detection and correction. Straddled area and dotted lines indicate the enhancements.

3. Error Types and Detection

Since we are going to change the voltage swing (i.e. overclock the register file), errors can occur during the writing of a register value or during the reading. In either case, the error(s) should be detected. So, all of the schemes we will discuss requires a detection mechanism. Figure 8 presents the register file and function unit segment of the architecture enhanced with the error detection and correction. The Error Detection & Correction (EDC) hardware stores the extra data bits and logic needed to perform the error detection and correction. During reading, the value from the register file is provided to the function units before it is checked. While the function units are operating, the error detection is performed. If an error is detected, the output of the function unit is omitted, the instruction is marked as *corrupted* and it is "replayed". Since the output of this instruction will not be written to the register file, all the dependent instructions will be replayed automatically. The original instruction that caused the error itself will be checked at the flags stage of the pipeline and replayed, because of the "*corrupted*" mark. Replay mechanisms have become an essential part of deeply-pipelined processors. In this scheme, the re-execution is initiated by the replay mechanism, which detects the instructions that do not receive correct input values (incorrect value can be caused by a cache miss) and re-executes them by informing the issue queue. As described above, we utilize the same hardware structures to re-execute the instructions that receive incorrect input values due to read or write errors. This way we can overlap the error detection with function unit computation and therefore push the detection circuit out of the critical path of the processor. Note that several processor architectures employ error detection and recovery schemes in their register files, e.g. IBM G5 uses an ECC-based scheme [20]. Therefore, the error detection required in our scheme would not incur an additional penalty.

An error detected during the reading will initiate an error check for the register value, because a read error might be caused by a write error (i.e. if the value written is incorrectly). During writes, we have to generate the detection bits in the EDC circuit. If the same register is accessed in the next cycle after write, we do not start the detection until the value is placed into the output. This gives us a one-cycle window after the write is completed. Therefore, the generation of the detection bits can be performed on the background in two cycles. However, the detection should be done in a single cycle (there can be single-cycle function units). Particularly, the detection should be done in the time the slowest function unit operation completes. Although this overlapping allows us to use EDC mechanisms off the critical path, we still cannot implement very complicated EDC mechanisms.

The errors during writes will be detected during reads as explained above. If an error is detected and can be corrected, the instruction will be replayed. If the error cannot be corrected, we use check-pointing techniques to restore the state of the processor to a correct one. We must,

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

however, note that in the experiments in Section 5, the probability of a rollback when the error correction schemes were utilized was always less than the probability of a rollback with the base architecture that do not utilize error recovery but allows full voltage swing. Hence, in practice we reduce the probability of system failures and rollbacks due to errors even if we increase the frequency.

4. Error Correction Schemes

We propose different error correction schemes to be used to increase the reliability of register files in the processors. First, we discuss the applicability of existing error correction/detection techniques in Section 4.1. Then, we propose redundancy-based schemes. We present a replication-based scheme (RP) in Section 4.3.

4.1 ECC-Based Schemes

There is a large space of possible implementations for error correction. Our framework can utilize any of these techniques. However, these techniques (such as Reed-Solomon or Hamming codes [6]) are usually computationally complex. Hence, they would not be able to capture the errors in the required time. As we have discussed in the last section, errors should be detected in a single processor cycle. For a 4 GHz processor, this corresponds to 0.25ns. To our best knowledge, none of the existing ECC techniques would be able to meet this time constraint. In the cases where the errors occur randomly, Hamming codes have been shown to be efficient to recover from the errors. Therefore, we consider them as an alternative error correction scheme. In our simulations, we use a code for detecting 2-bit errors and correcting single-bit errors. Since we simulate 64-bit registers, this requires 8 additional bits for each register.

Parity and ECC are two common alternatives for protecting register files against transient errors. Although a parity-based protection is not expensive to accommodate (from both performance and energy perspectives), it is limited since no error correction is provided. ECC schemes, on the other hand, can correct single or multiple bit errors. However, they incur high power consumption and latency overheads. Even a simple ECC scheme can take up to three times the delay of a simple ALU operation [25]. 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 [15]. Therefore, a scheme that provides correction with small energy and delay overhead is desirable.

4.2 Redundancy-Based Schemes

High performance processors aim to execute multiple instructions per cycle. One important obstacle to achieve this is the dependencies between instructions. Although RAW (reading a value after it has written) dependencies cannot be eliminated, register renaming is used to eliminate WAW (write after write) and WAR (write after read) dependencies. To perform register renaming, processors implement more physical registers than architectural registers. For example, Pentium 4 has 128 integer registers for 8 architectural (i.e. logical) registers [8]. Similarly, Alpha 21264 has 80 integer physical registers for 32 architectural register [10]. Then, for practically each destination register, register renaming maps the architectural destination register to one of the available physical registers. Thereby, if two instructions write to the same architectural register, they can still be executed in parallel because they will write their results to different physical registers. Regardless of the implementation for each instruction two tasks have to be performed to complete renaming. First a new register has to be allocated for destination register(s). Second, the source register(s) should be renamed such that they will be mapped to the correct physical registers. Figure 9 presents the register renaming implementation that is used in our experiments.

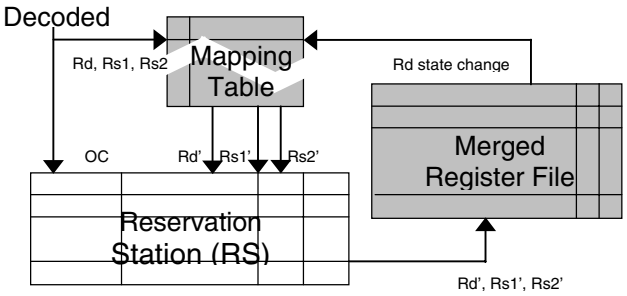

Figure 9. Physical structures associated with register renaming. Mapping table stores the architectural to physical register mappings.

The renaming scheme in our research is similar to the one used in Alpha and Pentium 4 (earlier Pentium architectures were implementing a Reorder Buffer). In this implementation, a mapping table keeps track of the physical registers that correspond to architectural registers. For example, if the architectural register r1 is mapped to physical register p5, the entry in mapping table that corresponds to r1 contains the number 5. In addition, mapping table keeps track of the states of the physical registers. During the renaming stage, only physical registers that are "free" should be allocated.

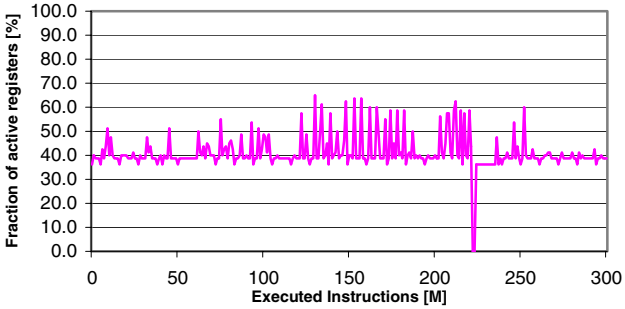

Figure 10. The fraction of active registers over the simulation of the 123.applu application.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

A physical register can be in one of the three states: available (AV) state, which indicates that they are not used; architectural register (AR) state, which indicates that the register is mapped to an architectural register; allocated, but not valid (AL) state, which indicates that the physical register is mapped to an architectural register, however the instruction that is generating the value is not completed yet. Once the instruction completes, the state is changed to AR. The registers in the AV state are free and can be mapped to architectural registers. In our techniques, we utilize these registers for copying the values of the active register (the ones in the AR and AL states).

4.2.1 OPPORTUNITY

Due to the nature of the applications and the limits on the Instruction-level parallelism (ILP) achieved, some of the physical registers may remain in the AV state for long periods of the execution. Our goal is to capture such periods and use these physical registers to store the copies of the active registers. Then, if an error is captured on the active register, we will use this copy value to restore the state of the processor. We first studied different applications for register usage, i.e. we studied the fraction of registers that are active during time epochs of execution of representative applications. The detailed simulation environment is explained in Section 5.1. Figure 10 presents the fraction of active registers over the simulation of a representative application. During several long periods of the simulation, more than half of the registers are not used. This indicates that a redundancy-based scheme can efficiently use these registers to store copies of active registers. We must note that the results presented here are for a representative application. Although a large fraction of the physical registers are not used during the execution of this application, the register file needs to be kept large enough for other applications, which might need all the available registers.

4.2.2 REDUNDANCY-BASE SCHEME (RB)

Redundancy-based (RB) scheme tries to allocate the free registers for copying the values of the active registers. Then, if an error is detected in the original value, this copy is used to restore the correct value. If the copy value is also corrupted, the error cannot be recovered. In RB, this allocation is performed during the register renaming stage. Specifically, during the register renaming, the register renaming logic additionally allocates a register that will be used for copying the register value. The copy register name is placed into the RUU (or the Reservation Station) along with the operation code and source and destination registers. At the completion of the instruction (i.e., when the value is written to the register file), the copy register is written.

We have to make three modifications to the traditional register renaming structures (depicted in Figure 9) to implement the RB. First, the mapping table is enhanced to select a copy register and store the selected copy register name. In addition to that, each physical register can be in an additional state called copy, which indicates that it is used as a copy register. Second, the Reservation Station (or RUU) is 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 Reservation Station should be modified to contain this information. Finally, we need to make a modification to the register file as well. It should be enhanced to perform the copy operation. Among the possible implementations, the simplest is to add a "copy" port for each write port in the register file. We only need to input the name of the copy register. The value of the copy register will be read from the corresponding data input for the write port and be written into the register name given in the copy port. Addition of the copy ports in the register file is likely to increase the latency of the register file. However, the copy port is easier to implement than a write port, because it does not require any additional data input.

During certain periods of the execution, the fraction of the active registers goes above 50%. This means that some registers will have no copies. If during the allocation of a copy register, there are no free registers (all the registers are in AR, AL, or copy states), the RB randomly selects one copy register and overwrites it with the new copy value. If there are no copy registers (all registers are in AR and AL states), the replication fails, i.e. no copy will be generated for the current destination register.

4.2.3 REDUNDANCY-ENHANCED SCHEME (RE)

The RB scheme has a significant drawback. In many cases, it might happen that a register can lose its copy before it is read. If there was an error during the write operation, the value cannot be recovered if the copy is overwritten. To address this shortcoming of RB, we implemented the Redundancy-Enhanced (RE) scheme.

RE scheme guarantees that if a copy value is overwritten, the corresponding active register does not have an error. To achieve this, the register renaming circuit introduces a dummy instruction to the pipeline that reads the value from the active register. For example, assume register p9 is storing the copy of the active register p3. Assume that p9 will be used to store copy of another register. The register renaming circuit puts a dummy instruction that reads the value of p3 into the pipeline. If during this dummy read operation an error is detected, the error detection and correction will be performed as usual. Hence, if there was an error during the write of p3, the error will be corrected. As we will show in Section 5, RE improves the performance of RB significantly. However, it can still not achieve 100% recovery because of failed copy attempts (~2%). Although this seems to be a small fraction, we have seen that it can have a significant impact on the recovery success.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

4.3 REPLICATION-BASED SCHEME (RP)

The last alternative we consider is called replication-based scheme (RP). In this scheme, we employ a second register file which snoops the writes to the primary register file and replicates all the values written to the primary register file. Specifically, the replica register file will store a value whenever a write operation to the primary register file occurs. Then, if during an access to a register, an error is detected, the replica register file will be accessed to retrieve the correct value. This correct value will be stored in the primary register file for further accesses to the data. The replica register file has a corresponding write port for each write port in the primary register file. On the other hand, the read ports of the replica file are only accessed when an error is captured in the primary register file. Hence, in our experiments we set the number of read ports in the replica file to 2. Note that, the area of the register file is dictated by the number of ports in it. Since in the replica file, the number of read ports is going to be smaller, the size of the replica register file will be smaller than the original register file. We believe that the overall complexity is tolerable because the register files usually consume a small fraction of the overall chip area. Note that the reads from the replica register file can be performed in multiple cycles. Therefore, the error rates during reads can be reduced. However, writes to the replica register file has to be completed in the same duration as the writes to the primary register file. Therefore, the probability of write errors remains the same for the replica file. This can be improved by having multiple replica files. However, such schemes are out of the scope of this paper.

We must note that an alternative scheme where we double the width of the register and write two copies at the same time can also provide a solution to the problem discussed in this paper. Then instead of using ECC or Parity, errors can be captured by comparing the two values. However, this requires a change in the main register file, which might degrade the overall performance. In the RP scheme, on the other hand, the duplicate register file occupies less space because of the smaller number of ports.

4.4 ALTERNATIVE IMPLEMENTATIONS

In our detection scheme, we assumed that any error is detected before the result of the operation is written back to the register file. If we allow the result to be written and "terminate" the instruction at a later stage in the pipeline (such as flags stage), we can utilize even more complicated schemes for detection. However, in such schemes, the rollback policy must be complicated to detect the instructions that have used this incorrect value, which will require significant modifications to the overall datapath design. Alternatively, the pipeline can be flushed to rid of all possible dependant instructions. However, in many configurations, the number of errors can be fairly high and flushing reduces the performance. Therefore, we do not consider such schemes.

One can imagine a scheme where only the values of the architectural registers are stored. Then, when an error is detected, the processor state is restored using this architectural register file. Similar to flushing, this technique has large impact on the performance and hence is not considered in this work.

5. EXPERIMENTS

5.1 EXPERIMENTAL SETUP

The SimpleScalar [4] version 3.0 simulator is used to evaluate the proposed techniques. The necessary modifications have been implemented to perform register renaming, error probabilities during read and write operations, and the proposed error correction strategies. We use parity detection for RB, RE, and RP schemes. As we have discussed in previous sections, the techniques make use the selective replay capabilities that exist in modern microprocessors. Therefore, we have made changes to SimpleScalar to simulate a realistically sized issue queue, to model the events in the issue queue in detail, and to simulate a realistic scheduler under selective replay.

Table 1. Simulated applications and important statistic: the number of write errors and read errors occurred when the cycle time is reduced to 20% of full voltage swing.

Appln	cycle [M]	DL1 acc. [M]	Reg. reads [M]	Reg. writes [M]	Write Errors [K]	Read Errors [K]
168.wupwise	260.1	93.4	550.82	284.55	40.9	43.7
171.swim	837.5	97.5	344.10	127.46	52.4	116.2
172.mgrid	492.9	109.8	285.96	48.28	58.8	62.3
173.applu	661.9	114.2	284.64	41.53	87.1	93.5
177.mesa	147.8	109.8	339.7	192.92	10.2	21.9
179.art	1845.7	102.8	309.8	125.65	87.1	218.4
183.equake	1407.6	127.2	436.50	183.93	145.7	189.3
188.ammp	762.8	116.2	501.86	195.35	37.8	91.7
189.lucas	567.2	72.0	338.17	154.46	60.7	78.3
301.apsi	308.6	111.8	571.27	230.48	34.2	41.1
FP. Average	729.2	105.5	396.28	158.46	61.5	95.6
164.gzip	200.8	71.8	480.1	309.7	27.3	44.5
175.vpr	682.3	118.8	428.2	248.9	72.8	99.4
176.gcc	376.0	126.7	459.7	270.5	17.4	43.9
181.mcf	2151.6	20.3	260.4	185.3	92.7	316.6
186.crafty	308.8	119.5	450.8	280.5	15.8	35.2
197.parser	576.8	89.2	498.1	289.8	38.1	73.2
253.perlbmk	261.5	108.3	419.3	240.4	19.4	41.0
254.gap	230.4	115.1	459.4	297.9	19.4	36.0
255.vortex	314.2	124.8	317.9	185.1	25.5	51.1
300.twolf	802.7	100.1	518.2	300.5	79.8	132.5
Int. Average	590.5	99.5	429.2	260.9	40.8	87.3
Average	**659.9**	**102.5**	**329.9**	**178.0**	**51.2**	**91.5**

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

We simulate an 8-way processor with a 16K, 4-way associative level 1 data cache, 16K, 2-way associative level 1 instruction cache, and 256K, 4-way associative level 2 cache. Level 1 caches have 2 cycle latencies and level 2 cache has 18 cycle latency. We simulate a register file similar to that of Alpha 21264 [10] with 80 floating point and 80 integer registers. Note that Alpha has 32 architectural floating point and integer registers. We used a bimodal branch predictor of size 4K. Our base processor has 20 pipeline stages with 7-cycle load loop (similar to the Pentium 4). Errors that cannot be recovered empties the pipeline and induces a 1000 cycle extra latency.

We simulate 10 floating-point and 10 integer benchmarks from the SPEC2000 benchmarking suite. The remaining benchmarks are not simulated due to the simulation problems we have encountered. We simulate 300 Million instructions after fast-forwarding an application-specific number of instructions as proposed by Sherwood et al. [22]. Detailed characteristics of the applications are presented in Table 1. However, in the rest of the paper, we do not present results for individual applications because their behavior is similar with respect to different configurations. Instead, we present the average results for all the simulated applications.

5.2 FIT Measurement

We analyzed the FIT behavior resulting from our schemes on the SPEC benchmark programs. We introduced faults in the register file guided by the fault occurrence probability obtained in equation (4). If the fault is not detected by the protection scheme (parity or ECC) it causes an application error.

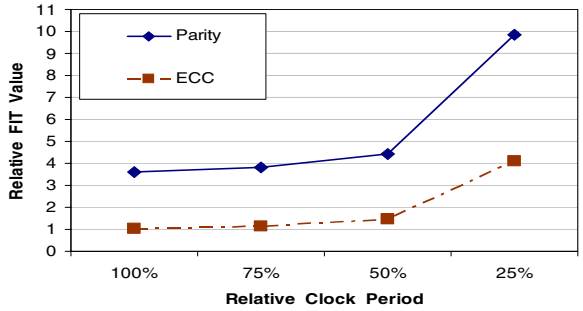

Figure 11. Increase in the FIT number while increasing the frequency.

Figure 11 presents the average relative FIT values under different relative clock frequency for parity and ecc-protected register file. The FIT value observed for ECC based system running in 100% clock frequency is considered as the baseline case. The relative FIT value is defined as the change in FIT in the corresponding case with respect to the FIT of the baseline processor. We see that reducing the cycle time by 25% (i.e., changing it from 100% to 75% relative clock cycle) has only a minor impact on the error behavior. Running the system in twice the original frequency (50% clock cycle time) causes the FIT numbers to increase by approximately 40% and 20% for ECC and parity protection schemes, respectively. Note that, executing a process with 25% relative clock cycle (increasing the clock frequency by four times) increases the FIT of the systems by approximately 200%. Our results are a good indication that the total number of failures due to our optimizations will be limited even for very aggressive overclocking of the register file.

5.3 Performance Optimization

We have performed two sets of experiments. In the first set, the designer is given a delay constraint for the register file. In these experiments, we are trying to measure the effects of introduced errors on the overall performance of a given architecture (the number of execution cycles). Hence, architectural parameters such as pipeline depth are kept constant. In the second set of experiments, we are given pipeline properties of a processor. We reduce the register file access times to reduce the corresponding number of pipeline stages.

Figure 12 summarizes the results for a fixed architecture. Each point in the figure corresponds to the average increase in execution cycles of 20 SPEC applications for the simulated scheme/frequency. We see that even with the simplest scheme (RB) we can reduce the cycle time by more than 60% while keeping the penalty under 1%. For ECC and RP, we can increase the frequency by 5 times while having 0.14% and 0.12% penalties.

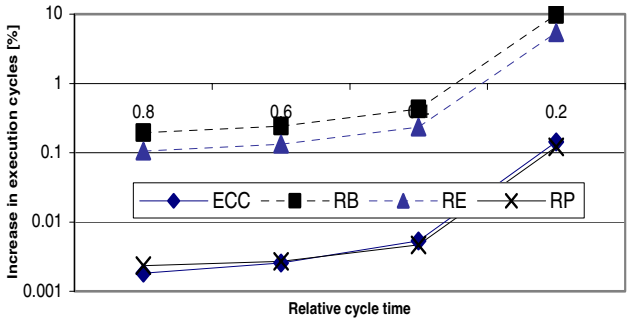

Figure 12. Increase in the execution cycles while increasing the frequency. Note that the y-axis is in logarithmic scale.

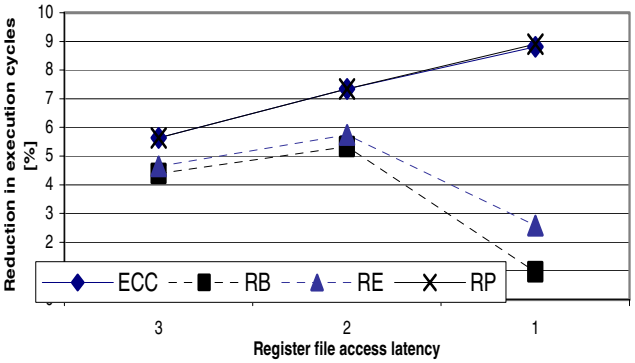

Figure 13. Reduction in average execution cycles for applications when varying the register file access latency between 4 and 1.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

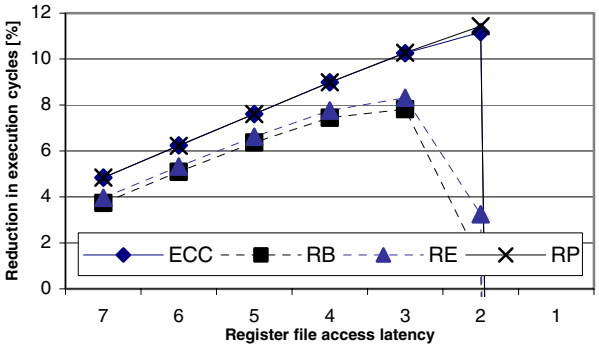

Figure 14. Reduction in average execution cycles for applications when varying the register file access latency between 8 and 1.

The second set of experiments is conducted for finding the optimal operation frequency given that the number of register file access pipeline stages varies with the frequency. As we reduce the access latency, the number of stages in the critical loops decreases. This increases the performance of the processor. However, since we introduce additional replays due to register read/write errors, there can be an increase in the execution cycles additionally. We performed simulations for the base architecture, a base architecture with 4-cycle register file access, and a base architecture with 8-cycle register file access2. Reducing the latency to 1 cycle from 2 results in 7.4%, 6.5%, 6.8%, and 7.5% reduction in execution cycles for the ECC, RB, RE, and RP schemes. The results for the processor with 4-cycle register access latency are summarized in Figure 13. Although processors such as Pentium 4 dedicate 2 pipeline stages for the register file access, this is likely to increase in the near future with the increase in the register file size and the overall number of pipeline stages. Figure 133 presents the average reduction in the execution cycles for 20 SPEC applications as we change the cycle time of the register file. We see that reducing the latency to 50% (i.e. to 2 cycles) has a positive effect for all recovery schemes. Reducing beyond this point, on the other hand, reduces the advantages seen by RB and RE schemes, where the number of rollbacks increases due to errors that cannot be recovered. RP and ECC, on the other hand, can recover from most errors. Therefore, they give their best performance for an access latency of 1. Specifically, ECC and RP reduce the number of execution cycles by 8.7% and 8.9%, respectively. The results for a processor with 8-cycle register access latency are presented in Figure 14. Similarly, as we reduce the latency, we generally see an increase in performance. However, when the access latency is set to 1, all techniques significantly increase the execution cycles: ECC, RB, RE, and RP increase the execution cycles by 26, 77, 69, and 24 times respectively.

There are two reasons for this. First, even if all the errors can be recovered, the processor spends most of its time replaying instructions due to register read errors. In addition, many times the errors cannot be recovered. Hence the rollbacks constitute a significant overhead. In fact, this is the only configuration in our simulations where the probability of a rollback is larger than a base architecture with full voltage swing. Overall, the RP gives the best result by reducing the execution cycles by 11.5% when the register file access latency is reduced to 2 cycles.

[2] Note that 2 pipeline stages are dedicated to register file access in Pentium 4. If the total number of pipeline stages is increased, the number of stages dedicated to register file accesses is likely to increase as well.

6. Related Work

Fault tolerant computing has been studied in detail in the context of high radiation environments and outer space [17, 26]. Techniques exist to study potential errors in the pre-silicon [2] stage and subsequent to the fabrication process [14]. More recently, designing computer systems for resiliency [12, 18, 19, 21, 27, 28] to transient faults has gained greater significance due the combined effect of higher integration densities, lower voltages, and faster clock frequencies. In comparison to our study, these techniques aim to increase the reliability of the processor with minimal impact on performance. Nakka [13] proposed RSE framework which provided reliability and security support. Bower [3] introduced SRAS which masked hard faults in microprocessor array structures. Both of these approaches have hardware and performance overheads. Our work, on the other hand, aims to increase the performance without affecting the overall reliability.

There is a recent trend in computer architecture to design processors that can adapt to circuit-level phenomena. Examples of this trend include Razor [4], thermal control schemes [24], and techniques for reducing inductive noise [16] and voltage variation [9]. Among these studies, Razor [5] is the closest work to ours. In Razor, the performance of the processor is reduced to achieve lower energy consumption by reducing the supply voltage in each pipeline stage. There are two major differences between Razor and our study: our goal is to improve the performance, whereas Razor improves the energy-efficiency while having a negative impact on the performance. Second, the particular technique we apply on the architecture is different. In short, to our best knowledge there is no work that studies the effects of operation frequency on reliability and trades off reliability for increasing performance, which is the focus of our paper.

7. Conclusions

In this paper, we have presented a method for reducing the cycle time of register files in high-performance microprocessors. We have first established a model for estimating the probability of a bit error when the cycle time of a register is reduced. When the cycle time is reduced, a

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

circuit node will experience reduced voltage swing, hence the probability of an error due to noise and other external factors increase. Then, we have presented novel architectural techniques to increase the robustness of the register file. Our goal is to allow the circuit designer to push the frequency higher (hence increase the probability of an error) and recover from these errors with the architectural techniques developed. We first showed a novel system for error detection and correction (EDC), which pushes the EDC logic out of the critical path of the processor. Then, we showed that a large fraction of physical registers are not utilized during certain periods of execution in superscalar processors. The redundancy-based schemes use these underutilized registers to copy the values of active registers. We discussed an Error-Correction Code (ECC) based on Hamming codes and a replication-based scheme, which uses a replica register file to store the copies of the active register values and uses those copies in case of errors to restore the state. Finally, we have presented experimental results showing that using the proposed techniques the frequency of the register file can be reduced by as much as 80% while having a 0.12% penalty in number of execution cycles. In addition, the number of pipeline stages in a processor with 4-cycle register file access can be reduced by 75%, resulting in a reduction of 8.9% in total execution cycles.

REFERENCES

[1]. Borch, E., et al. *Loose Loops Sink Chips*. in *International Conference on High Performance Computer Architecture (HPCA-02)*. Feb. 2002. Boston, MA.

[2]. Bose, P. *Ensuring dependable processor performance: an experience report on pre-silicon performance validation*. in *Intl Conference on Dependable Systems and Networks,* July 2000.

[3]. Bower Fred., et al. *Tolerating Hard Faults in Microprocessor Array Structures*. in *International Conference on Dependable Systems and Networks (DSN)*. June, 2004. Florance, Italy.

[4]. Burger, D. and T. Austin, *The SimpleScalar Tool Set, Version 2.0*. 1997, Univ. of Wisconsin-Madison, Comp. Sci. Dept.

[5]. Ernst, D., et al. *Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation*. in *International Symposium on Microarchitecture*. Dec. 2003.

[6]. Hamming, R.W., *Error detecting and correcting codes.* Bell Sys. Tech. Journal, 1950. **29**: p. 147-160.

[7]. Hartstein, A. and T.R. Puzak. *Optimum Pipeline Depth for a Microprocessor*. in *International Symposium on Computer Architecture*. May 2002. Anchorage / AK.

[8]. Hinton, G., et al., *The microarchitecture of the Pentium 4 processor*. 2001.

[9]. Joseph, R., D. Brooks, and M. Martonosi. *Control Techniques to Eliminate Voltage Emergencies in High Performance Processors*. in *International Symposium on High Performance Computer Architecture*. Feb. 2003.

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

[11]. Mallik, A. and G. Memik. *A Case for Clumsy Packet Processors*. in *International Symposium on Microarchitecture*. Dec. 2004. Portland, OR.

[12]. Mukherjee, S.S., M. Kontz, and S.K. Reinhardt. *Detailed Design and Evaluation of Redundant Multithreading Alternatives*. in *International Symposium on Computer Architecture (ISCA)*. May 2002.

[13]. Nakka N., et al. *An Architectural Framework for Providing Reliability and Security Support*. in *International Conf. on Dependable Systems and Networks (DSN)*. June 2004. Florence, Italy.

[14]. Paschalis, A., et al. *Deterministic Software-Based Self-Testing of Embedded Processor Core*. in *Design Automation and Test in Europe (DATE)*. March 2001.

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

[16]. Powell, M. and T.N. Vijaykumar. *Exploiting resonant behavior to reduce inductive noise*. in *31st Annual International Symposium on Computer Architecture (ISCA)*. June 2004. Munich, Germany.

[17]. Prager, K., et al. *A fault tolerant signal processing computer*. in *International Conference on Dependable Systems and Networks (DSN)*. June 2000.

[18]. 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.

[19]. Reinhardt, S.K. and S.S. Mukherjee. *Transient Fault Detection via Simultaneous Multithreading*. in *International Symposium on Computer Architecture (ISCA)*. June 2000.

[20]. Reinhardt, S.K. and S.S. Mukherjee. *Transient Fault Detection via Simultaneous Multithreading*. in *27th Annual International Symposium on Computer Architecture*. June 2000.

[21]. 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.

[22]. Sherwood, T., E. Perelman, and B. Calder. *Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications*. in *International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*. Sep. 2001. Barcelona, Spain.

[23]. Shivakumar, P., et al. *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*. in *International Conference on Dependable Systems and Networks (DSN)*. June 2002.

[24]. Skadron, K., M. Stan, and T. Abdelzaher. *Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management.* in *International Symposium on High-Performance Computer Architecture*. Feb. 2002.

[25]. Tremblay, M. and Y. Tamir. *Support for Fault Tolerance in VLSI Processors*. in *International Symposium on Circuits and Systems*. May 1989. Portland, Oregon.

[26]. 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.

[27]. Vijaykumar, T.N., I. Pomeranz, and K. Cheng. *Transient-Fault Recovery via Simultaneous Multithreading*. in *International Symposium on Computer Architecture (ISCA)*. May 2002.

[28]. Zhang W., G.S., Kandemir M., Sivasubramaniam A. *ICR: In-Cache Replication for Enhancing Data Cache Reliability*. in *Dependable Computing and Communication Symposium (DSN-03)*. 2003.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 **IEEE**