Superscalar Processor Performance Enhancement Through Reliable Dynamic Clock Frequency Tuning*

Viswanathan Subramanian, Mikel Bezdek, Naga D. Avirneni and Arun Somani
Dependable Computing and Networking Laboratory
Iowa State University, Ames, IA, USA
{visu, mbezdek, avirneni, arun}@iastate.edu

Abstract

Synchronous circuits are typically clocked considering worst case timing paths so that timing errors are avoided under all circumstances. In the case of a pipelined processor, this has special implications since the operating frequency of the entire pipeline is limited by the slowest stage. Our goal, in this paper, is to achieve higher performance in superscalar processors by dynamically varying the operating frequency during run time past worst case limits. The key objective is to see the effect of overclocking on superscalar processors for various benchmark applications, and analyze the associated overhead, in terms of extra hardware and error recovery penalty, when the clock frequency is adjusted dynamically. We tolerate timing errors occurring at speeds higher than what the circuit is designed to operate at by implementing an efficient error detection and recovery mechanism. We also study the limitations imposed by minimum path constraints on our technique. Experimental results show that an average performance gain up to 57% across all benchmark applications is achievable.

Keywords: Superscalar processor, Dynamic overclocking, Fault-Tolerant Computing, Reliability.

1. Introduction

The performance of processors has traditionally been characterized by their operating frequency. The operating frequency at which a processor or any digital system is marketed, is the frequency at which it is tested to operate reliably under adverse operating conditions. In order to satisfy timing criteria, designers are forced to assume worst case conditions while deciding the clock frequency. Such worst case timing delays occur rarely, allowing possible performance improvement through overclocking. Over the last decade, overclocking as a means to improve processor performance is gaining popularity [3]. Overclocking does not guarantee reliable execution. To reliably take advantage of this performance improvement, it is necessary to tolerate timing errors, when they occur.

The variables affecting propagation delay can be divided into physical variations (introduced during fabrication) and environmental variations (introduced during processor operation) [12]. Physical variations lead to both inter–die and intra–die variations. Inter–die variations are largely independent of design implementation. Intra–die variations, which are dependent on design implementation, are mostly caused by variations in gate dimension. To account for these variations, designers often assume delays three sigma from the typical delay. Environmental variations such as temperature and power supply voltage also have an effect on the delay through any path. These conditions can only be estimated when fixing the clock period of a circuit.

The worst case delay will be observed only if the longest path is exercised by the inputs. The input combinations which are responsible for the worst case path rarely occur. For example, in the case of a ripple–carry adder, the longest delay occurs only when a carry generated at the first bit position propagates through all remaining bit positions. However, a carry chain of this sort is very rare for both random and application generated input vectors [1].

Physical and environmental factors, along with the critical path of the design, force designers to opt for worst case clock periods to ensure error free operation. Since the clock period is fixed at much higher value than what is typically required, significant performance improvements can be achieved through overclocking.

A new and more conservative approach than overclocking seeks to exploit the performance gap left by worst case design parameters, while at the same time providing reliable execution. This approach, coined “better than worst case design” [1], uses principles from fault tolerance, employing some combination of spatial and temporal redundancy.

---

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

This paper presents a solution, which addresses the limitations imposed by worst case design, called SPRIT\textsuperscript{E}, or Superscalar PeRformance Improvement Through Tolerating Timing Errors [2]. The SPRIT\textsuperscript{E} framework allows the clock frequency of a superscalar processor to be dynamically tuned to its optimal value, beyond the worst case limit. Because the frequency is dynamically modified as the processor is running, variations in the environmental conditions, such as temperature and voltage, as well as variations present from fabrication, are automatically adjusted for. As frequency scales to higher values, timing errors will begin to occur. To prevent these errors from corrupting the execution of the processor, fault tolerance in the form of temporal redundancy is used. Specifically, pipeline stages are augmented with a local fault detection and recovery (LFDR) circuit.

The amount of frequency scaling is strongly influenced by the number of input combinations responsible for the longer timing-paths. As frequency is scaled higher dynamically, more number of input combinations would result in error. Each time an error occurs, additional time is required to recover from that error. We monitor the error rate during run time, and based on a set tolerable error rate that does not affect the performance, we adjust the clock frequency dynamically.

Another factor that influences frequency scaling is contamination delay of the circuit. Contamination delay is the minimum amount of time beginning from when the input to a logic becomes stable and valid to the time that the output of that logic begins to change. We explain in Section 4 how contamination delay limits frequency scaling. In Section 5, we explain how we can overcome this limitation, using CLA adders to illustrate our point.

To evaluate the SPRIT\textsuperscript{E} framework, several experiments were performed. First, to explore the possibilities of dynamic frequency scaling, an 18x18 multiplier was operated at varying frequencies, and the number of resultant timing errors was observed. Next, having developed a LFDR framework that shows an achievable frequency 44\% faster than the worst case level with the multiplier, the technique was applied to a superscalar processor. Using a superscalar processor synthesized in an FPGA, the frequency and application dependent timing error behavior was analyzed. Then with these results, the ability of the SPRIT\textsuperscript{E} methodology to provide performance improvement was determined for various error sampling implementations. For long term execution, on an average, all benchmarks show an achievable performance improvement of up to 57\% when continuous error sampling technique is implemented.

The rest of this paper is organized as follows: Section 2 provides a review of related literature. The error mitigation technique and global recovery in superscalar processors is described in Section 3. In Section 4, a description of the clocking system used to generate the dynamically modifiable clock is given. In Section 5, the effect of contamination delay on frequency scaling is studied. In Section 6, dynamic clock tuning methodology is presented along with three different error sampling techniques. Our experimental framework and results are presented in Section 7. Section 8 concludes the paper.

2. Related work

Since the traditional design methodology assumes that clock frequency is fixed at the worst case propagation delay, a large body of work exists to improve synchronous circuit performance without violating this assumption. Common techniques such as device scaling and deeper pipelining have been extensively used to increase processor performance. However, as observed in [6, 7, 18], there is an upper bound on the effectiveness of these techniques.

Several strategies have been proposed that apply fault tolerance to a processor with the goal of improving performance past worst case limits. Both the SSD [9] and DIVA [17] architectures apply fault tolerance in the form of a redundant processor. In these mechanisms, instructions are re-executed and checked. Therefore, timing errors may be allowed to occur in the main processor. However, the authors do not analyze the frequency dependent error behavior and thus do not quantify the amount of achievable performance gain. In [10], the issue, register renaming, and ALU logic of a superscalar processor are replaced with approximate versions that execute in half the time, but not necessarily correct. Two versions of the original ALU and register renaming logic are required to detect errors in the approximate versions. Thus this scheme has a high overhead.

TEATIME, proposed in [16], scales the frequency of a pipeline using dynamic error avoidance. However, this technique ignores the input dependence of the observed delay. Thus, it will stabilize on a frequency that is too conservative. The optimal operating frequency of a processor is dynamically achievable only when timing errors are detected and recovered from. TIMERTOL [15] design methodology uses an overclocked logic block with multiple safely clocked blocks of the same logic.

The RAZOR architecture [4, 5] uses temporal fault tolerance by replicating critical pipeline registers in order to dynamically scale voltage past its worst case limits. RAZOR achieves lower energy consumption by reducing supply voltage in each pipeline stage. Our goal is to allow faster execution for non worst case data by dynamically varying the operating frequency. In RAZOR, an internal core frequency generator is available which is capable of generating clocks at different frequencies, and the duration
of the positive clock phase is also configurable. However, the clock frequency is configured and fixed before program execution. We dynamically adjust the clock frequency to the optimal value during run time. This requires online error rate monitoring but offers higher improvement. In [11], the trade-off between reliability and performance is studied, and overclocking is used to improve the performance of register files. We use overclocking in critical pipeline stages to improve the performance of superscalar processor.

3. Timing error mitigation

To allow a superscalar processor to operate at frequencies past the worst case limit, SPRIT³E uses local fault detection and recovery, adding redundant registers between pipeline stages. Our timing error mitigation scheme is similar to the one used in Razor [5]. This scheme is most suitable to deal with multiple bidirectional (0 to 1 and 1 to 0) errors [13].

A diagram of the SPRIT³E technique applied to a superscalar processor is shown in Figure 1. The LFDR circuit is highlighted in the figure. The first register is clocked ambitiously at a frequency higher than that required for error free operation. The backup register is clocked in such a way that it is prevented from being affected by timing errors, and its output is considered “golden”. In Figure 1, Main Clock is the clock controlling synchronous operation of the pipeline, as would be present in an un–augmented pipeline. PS Clock, or Phase Shifted clock, has the same frequency as the Main Clock, but is phase shifted so that its rising edge occurs after the Main Clock. Operation of the LFDR circuit begins when the data from the pipeline logic, Data In, is stored in the main register at the rising edge of Main Clock. At this point, Data Out provides the stored value to the next stage, which begins to compute a result. Then the rising edge of PS Clock will cause the input to be stored in the backup register. By comparing the output of both registers, a timing error is detected and proper recovery steps are taken, if needed, to ensure correct operation of the pipeline. When an error is detected, the erroneous pipeline stage locally recovers by overwriting the data in the main register with the correct data in the backup register. This happens because of the multiplexer that selects between Data In and the data stored in the backup register.

The backup register is always provided with sufficient time to latch the data, hence it is free from metastability issues. However, when the main register is overclocked, the data and clock inputs may transition at the same time, resulting in metastability of the main register. To handle metastability issues, a metastability detector [5] is incorporated into the LFDR circuit. When metastability is detected in the main register, it is handled like a timing error, and the recovery mechanism is initiated.

3.1. Global error recovery

In addition to local recovery, action must be taken on a global scale as well to maintain correct execution of the pipeline in the event of a timing error. Global recovery is necessary to stall pipeline stages before the one in which error occurred, and to insert a bubble, so that subsequent pipeline stages are not affected by the error. In Figure 1, a delay register is shown between the output of the re–order buffer (ROB) and the commit stage. This register is necessary to prevent any erroneous value from being committed during the clock cycle needed for error detection, and to ensure that the architectural state of the processor remains correct. Four error locations are shown in Figure 1, denoted as IF error, ID error, FU error (n denoting nth functional block), and ROB error. Although the exact error recovery steps taken vary by location, in terms of the behavior of the error handler, these errors can be handled in a similar manner. The global recovery steps, explained below, are in addition to the local recovery that takes place at the erroneous pipeline stage.

When an error occurs in the instruction fetch stage, the instruction that was sent to the decode stage is reduced to a no–op. Additionally, the program counter is stalled for a cycle, so that following the correction of the error, the next instruction is fetched from the correct address. Finally, since the program counter is not updated, any branch or jump instructions attempting to write to PC during the stall cycle is stalled for a cycle. All other instructions in the pipeline are allowed to continue execution.

Any error occurring in the instruction decode and dispatch stage is propagated to both the ROB, and the allocated functional unit. In the ROB, the most recent entry is cleared by updating the pointer to the head of the buffer. When the next instruction is dispatched, it will overwrite the faulty instruction. To clear the functional unit, the global error handler maintains a record of the functional unit used by the dispatcher in the previous cycle. When an error in the dispatch stage is detected, that unit is cleared to prevent it from writing a wrong value to the ROB. Finally, the signal
notifying the instruction fetch stage of a successful dispatch is lowered to prevent the IF stage from fetching the next instruction during the error correction cycle.

An error in the execution of a functional unit stores an incorrect value in the ROB. Additionally, the incorrect value is forwarded to other functional units whose operands depend on the result of the faulty FU. In the ROB, the instruction is invalidated to prevent it from being committed. The functional units that have begun execution using the erroneous value are also stopped. This is accomplished by sending an error signal using the existing forwarding paths. Finally, the available signal of the faulty functional unit is lowered to prevent the next instruction from being dispatched to that FU.

An error in the ROB output is prevented from committing in the next cycle by the addition of the delay register mentioned previously. When an error is detected, the delay register is flushed to prevent a faulty commit. Also, the ROB is prevented from attempting to commit a new instruction in the next cycle. This is accomplished by manipulating the ready to commit signal from the commit unit.

![Figure 2. ALU error recovery](image)

Figure 2. ALU error recovery

Figure 2 shows the timing details of the global error recovery scheme when an error occurs in the ALU functional unit. A series of ALU operations is considered as this is the worst sequence for an error occurring in the ALU. If a different type of instruction is fetched following the ALU instruction causing a timing error, that instruction would be successfully dispatched to a different FU. In the figure, the add instruction completes error free and moves to the ROB. The sub instruction, however, does not stabilize before being captured by the main register in cycle 3. This is detected, and the ID stage is prevented from dispatching the or instruction, effectively stalling for one cycle. Additionally, the incorrect value sent to the ROB in cycle 3 is cleared.

The system shown in Figure 1 is simplistic in that it assumes only one clock cycle for each pipeline stage. The method of error recovery presented here is easily extensible to the superpipelined case. Another important consideration with this design is initialization of the pipeline. Error detection and recovery is triggered only after meaningful data is present in both the main and backup registers. Also, following a pipeline flush caused by branch mis-prediction, error detection at a stage is stalled until meaningful data again reaches the stage. The delay before beginning error detection varies between stages, and is accounted for in the design.

The area overhead for timing error detection is kept low by re–using the combinational logic which makes up the pipeline stages, and by duplicating only critical pipeline registers. Circuitry is also added to perform global error recovery, but this is modest as well, since the logic involved is not complex and re–uses already existing signals in the pipeline. Overall, SPRIT3E provides a viable means of tolerating timing errors.

4. Dynamic frequency scaling

To support the LFDR circuitry and maximize the performance of the pipeline, the main and phase shifted clocks must be carefully generated.

The timing error tolerance provided by the SPRIT3E hardware requires support from precise clock generation. Figure 3 shows three possible ways of generating the two clocks when the worst case propagation delay is 10 ns, and the contamination delay is 3 ns.

- In Case I, there is no frequency scaling, and the clock period of the Main clock is equal to the propagation delay. As a result, there is no need to phase shift the PS clock. The two clocks are identical in this case.

- In Case II, the frequency of the Main clock is scaled to 9 ns. To compensate for this reduced clock period, the PS clock is phase shifted by 1 ns, so that from the rising edge of the Main clock to the second rising edge of the PS clock, we have the full propagation delay of the logic circuit. Although there is a rising edge of PS clock 1 ns after the rising edge of Main clock, it will not corrupt the data to be stored in the redundant register as new inputs to the logic will take at least a time period equivalent to contamination delay to change the output.

- Case III shows the maximum possible frequency scaling. In this case, the clock period of the Main clock is 7 ns, and the phase shift of the PS clock is 3 ns. It is not possible to scale further because if the phase shift is increased beyond the contamination delay of the circuit, the redundant register may get incorrect result and cannot be considered “golden”.
Impact of error rate on performance. A factor that limits frequency scaling is error rate. As frequency is scaled higher, the number of input combinations that result in delays greater than the new clock period also increases. Each error takes additional cycles to recover. The impact of error rate is analyzed as follows:

- Let $t_{old}$ denote the original clock period.
- Let $t_{new}$ denote the clock period after frequency scaling.
- Let $t_{diff}$ be the time difference between the original clock period and the new clock period.

If a particular application takes $n$ clock cycles to execute, then the total execution time is reduced by $t_{diff} \times n$, if there is no error.

Let $S_e$ denote the fraction of clock cycles affected, by errors, due to scaling.

Let $k$ be the number of cycles needed to recover from an error.

To achieve any performance improvement, Equation 1 must be satisfied.

$$S_e \times n \times k \times t_{new} < t_{diff} \times n$$

$$S_e < \frac{t_{diff}}{t_{new} \times k}$$

According to Equation 2, for Case III in Figure 3, the error rate must be higher than 42% for this technique to yield no performance improvement when $k = 1$.

**Speedup calculation.** The overall speedup achievable using our technique is derived below and is given by Equation 5.

In a computation, it is possible that when the clock frequency is scaled, there is an increase in the total number of execution cycles. In a pipelined processor, when the processor accesses memory, the number of clock cycles taken for that memory operation increases when the frequency is scaled, if the clock frequency of the memory remains constant. Consider a processor whose clock period is 10 ns, and a memory access which takes 20 CPU cycles. If after scaling, the clock period is reduced to 5 ns, then the same memory access would take 40 CPU cycles.

Let $S_e$ denote the factor by which the number of cycles, taken to execute an application, increases because of scaling. Let $ex_{old}$ denote the old execution time. Let $ex_{new}$ denote the new execution time. Let $S_{ov}$ denote the overall speedup achieved.

$$ex_{old} = n \times t_{old}$$

$$ex_{new} = n \times t_{new} + S_e \times n \times t_{new} + S_e \times k \times n \times t_{new}$$

$$S_{ov} = \frac{ex_{old}}{ex_{new}} = \frac{t_{old}}{t_{new} \times (1 + S_e + k \times S_e)}$$

For Case III in Figure 3, if we consider $k$ to be 1, $S_e$ to be 10%, and $S_{ov}$ to be 10%, then we achieve an overall speedup of 1.19.

### 5. Managing contamination delay to increase phase shift

As explained in Section 4, the dependence of phase shift on contamination delay leads directly to the limitation of the frequency scaling. In general, the maximum improvement, dependent on the propagation delay, $t_{pd}$, and the contamination delay, $t_{cd}$, is given by $\frac{t_{cd}}{t_{pd}}$.

Since contamination delay limits performance improvement, it might be worthwhile to redesign the logic and increase the contamination delay. But increasing the contamination delay of a logic circuit without affecting its propagation delay is not a trivial issue [14]. At first glance, it might appear that adding delay by inserting buffers to the shortest paths will solve the problem. But delay of a circuit is strongly input dependent, and several inputs play a role in deciding the value of an output in a particular cycle.

To show that it is possible to increase contamination delay without affecting the propagation delay, we experimented on a CLA adder circuit. A 32–bit CLA adder circuit has a propagation delay of 3.99 ns, but an insignificant contamination delay of 0.06 ns, thus allowing almost no performance improvement using our technique. Our experiments indicate that by carefully studying the input–output relationship of a given circuit, it is possible to overcome the limitation imposed by contamination delay on our technique. The following case study presents our experiments and results we achieved for a CLA adder circuit.

**Case Study: Increasing contamination delay of CLA adder circuits.** Let us first consider an 8–bit CLA adder. The propagation delay of the circuit is estimated to be
1.06 ns, and the contamination delay, 0.06 ns. We synthesized the circuit using Cadence BuildGates Synthesis tool in Physically Knowledgeable Synthesis (PKS) mode. We used the 0.18 um Cadence Generic Standard Cell Library (GSCLib) for timing estimation.

For the 8-bit CLA adder, from timing reports, we observed that just about 20% of the paths have a delay more than 0.75 ns. Though this is highly motivating and provides a strong reason to apply our technique, a 0.06 ns contamination delay acts as a dampener and we risk incorrect operation if the clock period is reduced beyond 1 ns. To overcome the limitation imposed by the contamination delay, we increased the contamination delay without affecting the propagation delay of the circuit. After carefully studying the propagation delay pattern, we observed that it is possible to increase contamination delay by distributing the additional delay, either to the input side or the output side, or both. More importantly, the overall propagation delay remained unchanged. Figure 4 shows the new CLA adder circuit.

After adding delay values, the contamination delay of the circuit now is 0.37 ns, while the propagation delay remains unchanged at 1.06 ns. Now 31% of the timing paths have a delay value greater than 0.75 ns. Having a control over the increase in contamination delay gives us an advantage to tune the circuit’s frequency to the optimal value depending on the application and the frequency of occurrence of certain input combinations. Introducing delay to increase contamination delay increases the area of the circuit. Therefore, judiciously increasing contamination delay makes sure that the increase in area is kept minimal.

Table 1 provides all relevant details before and after adding contamination delay in 8-bit, 32-bit and 64-bit CLA adder circuits. The propagation delay, $t_{pd}$ and contamination delay, $t_{cd}$ are given in “ns”, and the area is given in “$\mu m^2$”. As we can see there is an increase in area after increasing contamination delay. Using slower buffers, the increase in area can be significantly reduced. The intention of our experiments is to demonstrate that contamination delay can be increased without affecting propagation delay for certain circuits. However, delay addition increases power consumption. For the 64-bit CLA adder, the total power before adding delay is 0.0144 mW, and the total power increases to 0.0222 mW after adding delays.

<table>
<thead>
<tr>
<th>Adder</th>
<th>Original</th>
<th>Delay Added</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t_{cd}$</td>
<td>$t_{pd}$</td>
</tr>
<tr>
<td>8-bit</td>
<td>0.06</td>
<td>1.06</td>
</tr>
<tr>
<td>32-bit</td>
<td>0.06</td>
<td>3.99</td>
</tr>
<tr>
<td>64-bit</td>
<td>0.06</td>
<td>7.89</td>
</tr>
</tbody>
</table>

6. Dynamic frequency tuning and error sampling techniques

The dynamically tuned frequency is achieved through the global feedback system pictured in Figure 5. Before operation begins, a small, non-zero, error rate is programmed as the set point. The clock controller is initialized with the worst case delay parameters of the pipeline. As stated above, the initial frequency of the clocks is the worst case propagation delay, and the PS clock begins with no phase shift. The clock generator block consists of a voltage controlled oscillator (VCO) in series with 2 digital clock managers (DCMs). The VCO is able to generate a variable frequency clock to meet the value given by the clock controller. The first DCM locks the output of the VCO to provide the Main clock to the pipeline. The second DCM provides a dynamically modifiable phase shift. It takes the Main clock as well as the value requested by the clock controller to generate the PS clock. Both DCMs provide a locked output as well, which is used to determine when the Main and PS clocks have regained stability. During the period in which the clocks are being adjusted, the pipeline must be stalled. To avoid a high overhead from frequent clock switching, the number of timing errors in the pipeline will be sampled at a large interval.
When considering different sampling methods, there is a trade off between the allowable sampling frequency and the number of bits needed to store the history of errors used to measure the error rate. The length of the error history should be long enough to accurately estimate the error rate. In the following discussion, a window of 100,000 processor cycles is used. Three sampling methods are considered: discrete, continuous, and semi-continuous.

In the discrete method, a single counter keeps the error history, incrementing every cycle in which an error occurs. When the window of 100,000 cycles passes, the counter is checked, and depending on the set point, the clock period is adjusted. The error counter is then cleared to count the errors occurring in the next window. The maximum size needed for the counter in the discrete case is 17 bits.

On the other side of the spectrum, the continuous method uses a sliding window of 100,000 cycles to maintain the history of errors. To implement this window, a 100,000 bit shift register is used, one bit for every cycle in the window. The counter is incremented or decremented, if the value shifted in is not same as the value shifted out. There are 100,000 bits needed for the shift register and 17 bits for the counter.

In order to obtain benefits similar to the continuous case, yet avoid its high overhead, a third, semi-continuous method is used. In this method, the error window is divided into 5 counters. Each counter maintains the total errors occurring in separate 20,000 cycles of the error history. The counters are used in a rotating fashion so that at every sampling, the oldest counter is cleared and begins counting. Each counter needs 15 bits, so for the 5 counters, 75 bits will be required.

7. Experimental results

To gauge the performance improvements provided by the SPRIT\textsuperscript{3}E framework, a sequence of experiments were performed. An initial study of a simple multiplier circuit established that significant room for improvement does indeed exist. From there, applications executing on a superscalar processor were analyzed, and the effects of augmenting the pipeline with SPRIT\textsuperscript{3}E were calculated.

As a first step in evaluating this technique, the frequency induced timing errors of a multiplier circuit are observed. In [13], multiplier circuit error rates are analyzed for both inter-die and intra-die variations by effectively altering logic delay via voltage control. We perform similar experiments, but analyze operating frequency induced timing errors. The circuit is implemented in a Xilinx XC2VP30 FPGA. A block diagram of the system is shown in Figure 6.

As presented in previous sections, the main and PS clocks operate at the same frequency, with a phase shift between them. However, in this experiment, the period of the clocks remains constant at the worst case delay. The phase shift of the PS clock latches the multiplier result in the early register after a delay. In operation, two linear feedback shift registers provide random inputs to the multiplier logic each main clock cycle. To minimize the routing delays, an 18x18 multiplier block embedded into the logic of the FPGA is used. The output is latched first by the early register, and a phase shift later by the main register. Error checking occurs at every cycle, and is pipelined to allow maximum shifting of the PS clock. A finite state machine (FSM) is used to enable the error counter for 10,000 cycles. To prevent the counter from counting errors that occurs when initializing the pipeline, the FSM begins enabling after 4 delay cycles have passed.

The worst case propagation delay of the synthesized circuit is estimated at 6.717 ns by the timing analyzer. To allow plenty of time for the circuit to execute before being captured in the main register, a clock period of 8 ns is used. The phase shift of the PS clock is varied from 0 to -5.5 ns, giving effective clock periods of 8 to 2.5 ns. For each effective period, the total errors are counted for an execution run of 10,000 cycles. For instance, when the PS clock is shifted such that its rising edge occurs 5 ns before the main clock, the multiplier logic is effectively being given 3 ns to compute. At this frequency, about 94% of the 10,000 cycles produce a timing error.
Figure 7 presents the percentage of cycles that produce an error for different effective clock periods. As shown, although the worst case delay was estimated at 6.717 ns, the first timing errors do not begin occurring until a period of under 4 ns. Using a method such as LFDR to tolerate a small amount of timing errors allows this circuit to run at almost half the period giving a speedup of 44%.

![Figure 7. Percent of error cycles versus the clock period for the multiplier circuit and DLX processor](image)

### 7.1. Evaluation of SPRIT$^3$E framework

The SPRIT$^3$E framework is evaluated on a DLX superscalar processor [8]. The relevant parameters of the processor are summarized in Table 2. The superscalar DLX processor is synthesized for the Xilinx XC2VP30 FPGA. The maximum timing delay between registers in this circuit is 21.982 ns, between the source registers of the MDU and the data registers of the ROB. Similar delays, all around 20 ns, exist through the other functional units to the ROB, as well from the dispatch stage to the ROB. Thus, to analyze the timing error rates of the processor, the ROB registers are augmented with additional registers as well as the comparison and counting circuitry shown in Figure 6. The processor is operated at varying phase shifts of the PS clock, and the percentage of cycles in which an error occurred for the execution run is recorded. For the benchmarks run on the FPGA, the processor state and the output of the program is checked for correctness after program execution.

![Figure 7. Percent of error cycles versus the clock period for the DLX processor](image)

#### Table 2. DLX processor parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode / Issue / Commit bandwidth</td>
<td>2</td>
</tr>
<tr>
<td>Reorder Buffer Entries</td>
<td>5</td>
</tr>
<tr>
<td>Number of Arithmetic Logic Unit (ALU)</td>
<td>1</td>
</tr>
<tr>
<td>Number of Multiply Divide Unit (MDU)</td>
<td>1</td>
</tr>
<tr>
<td>Function Branch Resolve Unit (BRU)</td>
<td>1</td>
</tr>
<tr>
<td>Units Load Store Unit (LSU)</td>
<td>1</td>
</tr>
<tr>
<td>Instruction and Data Cache Size(Bytes)</td>
<td>64</td>
</tr>
<tr>
<td>Memory Size (KBytes) - 2 Cycle Access</td>
<td>64</td>
</tr>
</tbody>
</table>

Figure 7 shows the error rates of operating the DLX at effective periods between 10 and 3.5 ns for 3 different benchmarks. The RandGen application performs a simple random number generation to give a number between 0 and 255. One million random numbers are generated, and the distribution of the random variable is kept in memory. The MatrixMult application multiplies two 50x50 integer matrices and stores the result into memory. The BubbleSort program performs a bubblesort on 5,000 half–word variables. For this application, the input is given in the worst case unsorted order. As shown in the figure, for both RandGen and MatrixMult, the errors become significant at around 8.5 ns, while the error rate of BubbleSort stays low until around 8 ns. This is because both the MatrixMult and RandGen applications use the MDU, and thus are likely to incur the worst case path. The BubbleSort uses only the ALU to perform comparisons as well as addition and subtraction, so it is able to operate at lower periods before errors begin to occur.

Using the probability distribution for the error rate determined in the previous step, a simulator is written to evaluate the effectiveness of the SPRIT$^3$E framework using the different methods of error sampling discussed in Section 6 at a set tolerable error rate of 1%. In this experiment, the amount by which the clock period is allowed to change is held constant for different sampling methods. Each benchmark is evaluated separately, and is executed for its original number of cycles, reported in Table 3, as well as for a long run of 120 million cycles.

#### Table 3. Length of Applications in Cycles

<table>
<thead>
<tr>
<th>Application</th>
<th>Cycles to Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMult</td>
<td>2901432</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>118896117</td>
</tr>
<tr>
<td>RandGen</td>
<td>15750067</td>
</tr>
</tbody>
</table>

The scaling behavior for the matrix multiplier application executed for a long run is shown in Figure 8. The figure highlights the differences between discrete, semi-continuous, and continuous sampling. The other applica-
tions show similar period scaling over the course of execution. As the figure demonstrates, the long intervals between switching for the discrete sampling method prevent it from reaching the optimal period as quickly as the continuous and semi-continuous cases.

![Figure 8. Dynamically scaled clock period versus the elapsed cycles for MatrixMult](image)

As the simulation is running, the execution time of the application is calculated. The reference execution run sets the period at the worst case value and allows no scaling. Thus no timing errors occur. For the other cases, each cycle in which a timing error occurs results in a stall cycle being injected into the pipeline. Also, when a change in period occurs, the time taken to lock the DCMs to the new frequency is added to the total execution time. The execution times for each application when run for its original execution cycles is shown in Figure 9, normalized to the reference worst case time. The BubbleSort application shows the best performance as it runs the longest and thus runs the longest at the optimal period for any sampling method. The MatrixMult application, however, is only long enough for gains achieved by lowering the period to begin to outweigh the penalties for doing so. Each benchmark was also evaluated by running for a longer execution time. The performance results are presented in Figure 9. For this variation, all benchmarks perform similarly, with the discrete error sampling method giving on average a 43% improvement over the worst case, and the semi-continuous and continuous methods outperforming it at 56% and 57% respectively.

**Speedup Calculation:** Table 4 provides the speedup achievable for the multiplier circuit, the 32-bit and 64-bit CLA adder circuits, and the three different benchmarks run on the DLX processor augmented with the SPRIT3E framework. The overclocking technique is applied to the CLA adders after increasing their contamination delay, as explained in Section 5. The new contamination delay values are the ones reported in Table 1 in Section 5. The experimental setup for the adder is similar to the one explained for the multiplier circuit. Equation 5, derived in Section 4, gives the speedup, $S_{ov}$. We consider $k$ to be 1, and $S_c$ to be 10%. The $S_c$ factor is ignored for the multiplier and adder circuits. We calculate speedup for an error rate target of 1%.

<table>
<thead>
<tr>
<th>ALU Circuit</th>
<th>$t_{old}$</th>
<th>$t_{new}$</th>
<th>$S_{ov}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>6.72</td>
<td>3.75</td>
<td>1.77</td>
</tr>
<tr>
<td>32-bit CLA</td>
<td>3.99</td>
<td>3.3</td>
<td>1.19</td>
</tr>
<tr>
<td>64-bit CLA</td>
<td>7.89</td>
<td>6.2</td>
<td>1.27</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>21.98</td>
<td>8.5</td>
<td>2.33</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>21.98</td>
<td>8.0</td>
<td>2.47</td>
</tr>
<tr>
<td>RandGen</td>
<td>21.98</td>
<td>8.5</td>
<td>2.33</td>
</tr>
</tbody>
</table>

While calculating the speedup in Table 4, we did not take into account the time margins added to the propagation delay because of physical and environmental factors. In reality, the original clock period would be fixed at a higher value than the circuit’s propagation delay, leaving room for further frequency scaling.

**Impact on area and power consumption:** To guarantee reliable execution when operating at higher than worst case speeds, we introduced LFDR circuits in place of flip-flops in the pipeline stages, and to remove the limitations imposed by short paths on frequency scaling, we added delay buffers to increase the delay of short paths. This increased the area and power consumption of the superscalar processor.
Table 5 provides synthesis results for the unmodified DLX superscalar processor and the one augmented with the SPRIT3E framework. Both designs are mapped to Xilinx Virtex II Pro FPGA using Xilinx ISE 8.2 synthesis tool. There is a 3.12% increase in the number of flip-flops. The increase in the combinational logic part is 0.3%. The net increase in area because of the SPRIT3E framework is 3.2% (calculated from equivalent gate count). For our experiments on DLX superscalar processor, we did not increase the contamination delay of any pipeline stage.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Flip-flop Count</th>
<th>Comb. Area (4-LUTs)</th>
<th>Equiv. Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmodified DLX</td>
<td>5315</td>
<td>14363</td>
<td>164760</td>
</tr>
<tr>
<td>SPRIT3E DLX</td>
<td>5313</td>
<td>14407</td>
<td>170048</td>
</tr>
</tbody>
</table>

To minimize the increase in power consumption, we replaced only those pipeline registers in the critical path with LFDR circuits. From Xilinx Xpower reports, we observed that there is no significant difference in the total power consumed by the two processors.

8. Conclusions

As demonstrated by the successful timing error tolerant overclocking methodology, the current way of estimating the operating frequency for synchronous circuits is far too conservative. The SPRIT3E framework reuses existing superscalar pipeline logic whenever possible, resulting in a modest error detection and recovery logic overhead. Although our experiments are based on FPGA platform, an extension can be made to logic implemented in ASIC technology. This work presents an initial exploration of the possibilities for taking advantage of the margins produced by worst case design mentality. In the future, implementing a main memory system for the synthesized DLX processor would allow full scale benchmarks to be evaluated, as well as allow an exploration of the effect of increasing the clock frequency on the average instructions committed per clock cycle. Another important concern in using the SPRIT3E framework is how well the phase shift can be adjusted at high frequencies. This paper presents a very promising technique, with many exciting directions for the future.

References


