A Case for Clumsy Packet Processors

Arindam Mallik and Gokhan Memik
Department of Electrical and Computer Engineering, Northwestern University
{arindam, memik}@ece.northwestern.edu

Abstract

Hardware faults can occur in any computer system. Although faults cannot be tolerated for most systems (e.g., servers or desktop processors), many applications (e.g., networking applications) provide robustness in software. However, processors do not utilize this resiliency, i.e., regardless of the application at hand, a processor is expected to operate completely fault-free. In this paper, we will question this traditional approach of complete correctness and investigate possible performance and energy optimizations when this correctness constraint is released. We first develop a realistic model that estimates the change in the fault rates according to the clock frequency of the cache. Then, we present a scheme that dynamically adjusts the clock frequency of the data caches to achieve the desired optimization goal, e.g., reduced energy or reduced access latency. Finally, we present simulation results investigating the optimal operation frequency of the data caches, where reliability is compromised in exchange of reduced energy and increased performance. Our simulation results indicate that the clock frequency of the data caches can be increased as much as 4 times without incurring a major penalty on the reliability. This also results in 41% reduction in the energy consumed in the data caches and a 24% reduction in the energy-delay-fallibility product.

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 push the limits of performance of microprocessors. This is achieved through optimizations at the architectural 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 has become an important obstacle. Increased heat dissipation and sub-micron effects are two examples of the limitations on the optimizations at the circuit level. In this work, we design a micro-architectural optimization to aid the circuit designers to overcome such hurdles. Particularly, we will allow the clock frequency of the data cache to go beyond the specifications of the circuit designer. Instead of performing this “over-clocking” uninformed, we will first explore the relation between the operating frequency (i.e., clock frequency) of a cache structure and its robustness. As we increase the clock frequency, the probability of a fault in the data cache accesses increases. This may result an erroneous execution of the applications. Hence, we name our proposed architecture a *clumsy packet processor*. In our approach, we first develop a model for estimating the hardware faults when the clock frequency is changed. This model will allow us to develop ultra-low power cache structures. In addition, the delay of the components will also be reduced. The disadvantage of this optimization is that the probability of hardware failure reduces the reliability of the processor. Overall, our goal is to investigate the trade-offs at the application-level, architecture-level, and circuits simultaneously in the context of *packet processors*. We use the term packet processor for any type of processor handling packets in a networking hardware. These range from network processors (*NPs*) to ASICs and general-purpose microprocessors used in networking hardware.

In all computer processors there is an inherent possibility of faults[1] being introduced into the system. These faults may arise from any of several sources such as adverse environmental conditions [26], physical hardware defects, electronic noise or logical design flaws [9]. Moreover, this fault problem is expected to be even more pressing in the future due to aggressive scaling-down of the supply voltages (V_{dd}), increasing clock rates, and the use of flip-chip packaging. While it is critical to put every effort to avoid these faults by careful circuit design and packaging, they can still occur and need to be addressed. Hence, we should consider reliability trade-offs even during the design of the processors, which will operate completely under the specified conditions.

The effect a fault has on a system is largely dependent on the application in question. In most cases, omitting faults is not an option, i.e., the processor should be designed to capture and eliminate faults. This is the inherent nature of the user expectation. However, for other domains—such as networking and media applications—a certain level of error is acceptable, and the integrity of the system’s behavior can be maintained despite potential faults. This is also related to the properties of the systems: networking software/systems are implemented with the assumption that the hardware can fail (e.g., routers can drop packets).

Regardless of a fault’s source, the system will operate differently depending on the corrupted data. Electronic noise may lead to the corruption of a single piece of

[1] A fault is an incorrect execution of the hardware. An error is defined to be an incorrect outcome of an application due to a fault.

transient data and affect behavior only momentarily. On the other hand, a static data element might be damaged—such as a lookup table—disrupting the system for a longer period of time and perhaps making recovery from the error more difficult. In this paper we analyze the susceptibility of a data cache to faults and the resulting behavior for packet processors. Particularly, we study several networking applications and define error metrics for each of these applications. We first make the distinction between the control plane and data plane tasks in these applications and measure the error behavior of the applications under different operation frequencies in these segments. Then, we perform a study where we introduce cache faults and measure their effect on these applications. Our goal is to extract optimal execution properties of the caches for different applications. We also present a scheme that dynamically adjusts the processor properties to achieve reduced energy consumption and/or increased performance. Specifically, our contributions in this paper are:

- We propose the design and utilization of clumsy packet processors,
- We find a realistic model that determines the probability of a fault for a given cycle time of a cache and show that the delay of the cache and the energy consumed by the cache can be reduced significantly without incurring a large penalty on faulty behavior,
- We discuss simulation results investigating an optimal point for trading off the reliability for reducing cycle time of the data cache in a representative architecture,
- We implement a scheme to dynamically adjust the operation frequency of the data cache to achieve the desired objective (e.g., reduced energy).

There is also an increasing motivation to utilize NPs in wireless systems. In such systems, energy consumption is arguably the most important design criteria. Our optimization scheme reduces the execution delay and the energy consumption simultaneously.

The types of errors examined are similar to those in the aforementioned cases. One type is considered to be a *volatile* error, affecting data only temporarily. In general this type of error will only concern a limited amount of data, and will not noticeably affect performance provided that the error does not continually reoccur. The other type is a *nonvolatile* error, which has an effect on a static data structure (e.g., the routing table). This type of error will have a lasting effect on the system. Our goal in this paper is to define data structures in these applications that can be used to measure their error behavior.

The rest of the paper is organized as follows. In the next section, we present the applications studied and discuss the application-specific error metrics we have defined for each. Section 3 explains the analytical model we have developed to estimate the hardware faults for various clock frequencies of the data cache. Section 4 presents the overall architecture and a dynamic scheme for adjusting the clock frequency. Section 5 discusses the simulation results. Section 6 gives an overview of the related work and Section 7 concludes the paper with a summary.

2. Applications and error measurement

In this section, we discuss the networking applications studied in this paper and present the error metrics used for each application. We selected seven applications from the NetBench [13] suite. The applications are listed in Table I. NetBench is a benchmarking suite designed for NPs. It contains applications representing level 3 tasks (e.g., route) as well as higher-level programs.

As a metric of "reliability", we first identify important data structures and outputs of key function units for each application. Our goal is to make a comparison of these data values between the correct execution and an execution with faults (Section 5.2). Thereby, we will measure the probability of an error in the application. Some of these data structures have more impact on the overall output than others (e.g., a routing table error is more important than an error in the ttl value calculation). However, in this study we do not assign weights to them. Note that this type of measurement assumes that the application executes to completion even under faults. However, we are executing erroneous code (i.e., a code that will read erroneous data). As the data values are changed, it is possible that the application might fall into an infinite loop or even cause the system to crash. This is of interest to us for measuring the effects of faults. Therefore, an error, which prevents a complete execution is a special one called a ***fatal error***.

Table I. Networking Applications and Their Properties

App.	No. of inst. simulated [M]	No. of cache acc. [M]	Cache miss rate [%]	Fallibility Factor	
				C_r=0.5	C_r=0.25
crc	145.8	59.8	1.2	1.007	1.052
tl	6.9	3.9	9.2	1.016	1.135
route	14.2	7.1	5.8	1.001	1.018
drr	12.9	7.9	5.7	1.002	1.008
nat	11.4	5.6	7.1	1.004	1.077
md5	209.1	73.2	3.8	1.055	1.261
url	497.0	249.1	11.2	1.003	1.018

One common property of all the applications is the separation of control plane and data plane tasks. In all the applications, the implementation initially performs the control plane tasks. This is followed by the data plane tasks. We have identified each of these segments in the applications.

CRC: The CRC-32 checksum calculates a checksum based on a cyclic redundancy check. The code is available in the public domain [6]. The errors are measured using two data structures: the crc table and the crc accumulator value calculated for each packet. Note that the errors in the crc table are more serious, because they can potentially affect multiple packets.

TL: TL is the table lookup routine common to all routing processes. In this application, a radix-tree routing table is implemented. The code segment is from FreeBSD operating system [18]. The data values in the TL application are the radix tree nodes traversed and the RouteTable entry for each packet.

ROUTE: Route implements IPv4 routing according to RFC 1812 [4]. When a router receives a packet, it has to decide the next network hop. The values observed in the route application are the entries in the created RouteTable, the checksum value, the ttl value, and the radix tree entries traversed for each packet.

DRR: Deficit-round robin (DRR) scheduling [24] is a scheduling method implemented in several switches today. In DRR, all the connections through the router have separate queues for a fair scheduling. The implementation is based on the algorithm by Shreedhar and Varghese [24]. The data values in the DRR application are the entries in the created RouteTable, the radix tree entries traversed for each packet, the value of the deficit list for each packet, and the deficit information read for the packet.

NAT: Network Address Translation (NAT) is a common method for IP address management. NAT operates on a router, usually connecting two networks, and translating the private (not globally unique) addresses in the internal network into legal addresses before packets are forwarded onto the public network. The data values used for measuring errors in NAT are initial IP source address, value in the interface for translation, translated IP source address, the IP destination address after translation, the entries in the NAT table, and the radix tree entries traversed for each packet.

MD5: Message Digest algorithm (MD5) creates a signature for each outgoing packet, which is checked at the destination [21]. The implementation is from RSA Data Security, Inc. [22]. The errors in MD5 are binary errors

URL: URL implements URL-based destination switching, which is a commonly used content-based load balancing mechanism. In URL-based switching, all the incoming packets to a switch are parsed and forwarded according to URL. The data structures in the URL application that are observed are: URL table entries, final IP destination address, RouteTable entries, the checksum value, the ttl value, and the radix tree entries traversed for each packet

3. Fault model and clock 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, and may lead to different kinds of unexpected behavior including functional failure or logic faults. The parameters that determine if there will be a logic fault 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. Higher clock rates limit the achievable voltage swing at a circuit node (see Figure 1(a)), since there is insufficient time to fully charge or discharge the load capacitance. C_{fs} in Figure 1(a) is 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}.***

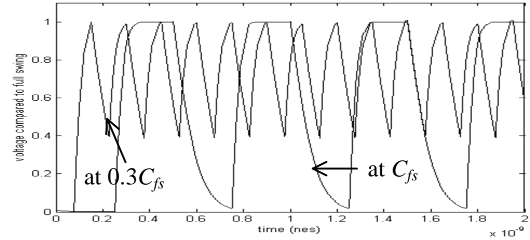


Figure1(a). Voltage at circuit node;

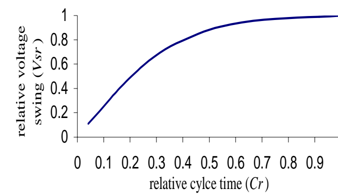


Figure1(b). Voltage swing-frequency curve

Figure 1(b) illustrates the decrease of voltage swing (V_s) 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. This shape correctly maps the change of actual signals on-chip with time. The curve in Figure 1(b) has been produced by simulating a chain of gates driven by an inverter at different frequencies with constant supply voltage V_{dd}.

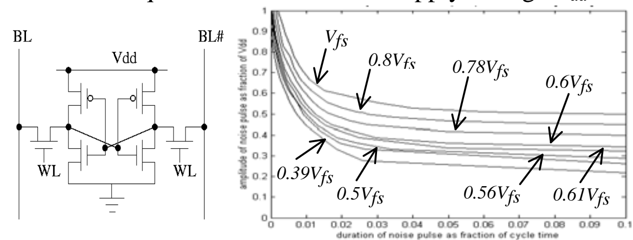


Figure 2 (a) A static RAM cell; (b) Noise Immunity Curve

With a reduced signal level, a circuit node is more likely to suffer from logic failure due to a certain level of noise. The main advantage of static logic over dynamic logic is its robustness under the influence of noise. But static logic may also suffer from logic failure if there is a feedback loop. A 6-transistor static RAM cell (as in Figure 2(a)), which is a common building block of caches, has a feedback loop that cannot recover from noise-induced faults. In these types of circuits there are three possible points where noise can be injected: the input, the clock and the feedback loop. Of these three points, the feedback loop is the most sensitive to noise. A set of noise immunity

curves for the SRAM cell in Figure 2(a) is presented in Figure 2(b), which plots the relative noise duration (D_r) against the relative noise amplitude(A_r) at various voltage swings. 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 2(b) represents the amplitudes and durations of a noise pulse that can cause logic failure. 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.

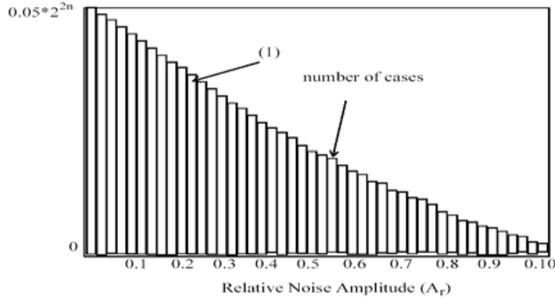


Figure 3. Noise level at various switching combinations.

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. 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 very small amplitude of noise is very 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 3. This distribution can be approximated very well by an exponential as in (1).

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

The exact constant K_1 and K_2 depends 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

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

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

The probability distribution of noise duration can be given by (3). The reason 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. 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 (P_E) of logic failure for an SRAM cell at different voltage swings have been obtained by the integration of the probabilities of noise pulse above each curve of Figure 2(b). Figure 4 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].

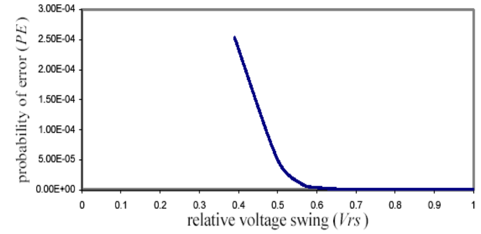


Figure 4. Probability of a fault at various voltage levels

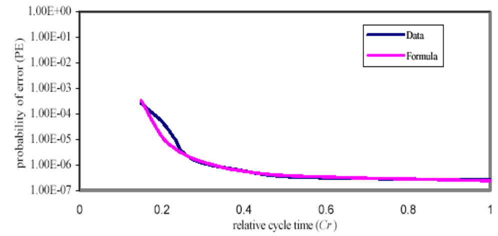


Figure 5. Probability of a fault at different cycle times

The probability of fault versus cycle time in Figure 5 has been obtained by the voltage swing variable from the two relations: cycle time versus voltage swing (Figure 1(b)) and probability of fault versus voltage swing (Figure 4). The relative cycle time C_r 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. P_E is a single bit probability of fault 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 P_E and C_r and F_r.

$$P_E = 2.59 * 10^{-7} * e^{\frac{1}{6 * C_r^2}} = 2.59 * 10^{-7} * e^{\frac{F_r^2}{6}} \tag{4}$$

These formulae have been found by curve fitting for the data of the above curves. The curves in Figure 5, showing the data and the curve fitted formula, illustrate the accuracy of the formula.

4. Clock variation and fault detection

In this paper, we assume a processor architecture similar to a generic Network Processor (NP). We model a relatively simple execution core with a local instruction cache, a local data cache, and a shared level-2 cache. Although we apply our ideas to a packet processor, they

can be applied to any type of processor that executes applications with fault resiliency (e.g., media processors).

One important aspect of the cache accesses is whether to include a fault detection scheme or not. In Section 5, we will experiment with a processor architecture where cache blocks are protected with parity and a processor architecture without any fault detection scheme. We are modifying the clock frequency of the level-1 cache only. Hence, we assume that the data in the level-2 cache will be correct unless an incorrect value from level-1 is written to it. Therefore, if a fault is detected, we can access the data from the level 2 cache. As the error correction techniques (such as Hamming codes) would incur unnecessary complication on the design and energy consumption, they are not considered in our studies.

Once a fault is detected, we have different options of recovery. A fault might be caused during the read—in which case the actual data in the cache is actually correct—or during the write to the cache . We cannot determine the exact source of the fault. The first technique we utilize assumes that every fault observed is a write fault. Therefore, for every fault detected, it invalidates the cache block[2] and starts accessing the level 2 cache. This strategy is called a ***one-strike*** strategy. The second strategy accesses the cache after a fault and if another fault is detected, it invalidates the cache block and accesses the level 2 cache. This strategy is called a ***two-strike*** strategy. Similarly, a ***three-strike*** strategy accesses the level 1 cache twice before invalidating the block. Even if the processor employs a fault detection mechanism, there is still a chance of faults. Therefore, the application can behave erroneously.

Over-clocking the cache can be utilized during the design process of a processor. However, this is hard to achieve for programmable processors (such as Network Processors), because different applications might require different levels of reliability. Therefore, in the next section we also present results for a ***dynamic frequency adaptation*** technique. In this scheme, the processor adapts the operation frequency of the data cache according to the faults it has observed. Particularly, it records the number of parity failures during execution epochs. For our simulations, after the completion of the processing of 100 packets, the processor makes a decision for whether to increase the frequency, to keep it in its current state, or to decrease it depending on the number of faults. Note that the possible frequency settings are discrete. Hence, when the frequency is changed, it will be set to the next frequency level available. Whenever a frequency change is made, the number of faults in the previous epoch is stored. During the decision, if the number of faults is more than X_1% of the last stored fault rate, the frequency is reduced. If the fault rate is less than X_2% of the last stored rate, the frequency is increased. For all other rates, the frequency is not changed. A detailed study reveals that setting X_1 to 200% and X_2 to 80% overall results in the best performance of the dynamic scheme. This also relates to the fault model we have developed in Section 3. As shown in Figure 5, the clock cycle can be reduced by almost 60% before we observe a major increase in the number of faults. Depending on the packet processing time, the X_1 and X_2 values will lean towards increasing the frequency until a significant increase in the number of faults.

Most networking applications have application errors proportional to the number of faults occurred during the processing of a packet. The dynamic frequency adaptation technique observes the packet processing and makes the decisions for a constant number of packets (instead of time). This allows the system to dynamically adjust to the properties of the application. This information is usually available to the cores.

Note that dynamically varying the clock frequency of the cache is easier to implement than varying the supply voltage [11]. This can be achieved while the cache is being accessed and there is no need to flush the cache. In accordance with this, we incur a 10-cycle penalty whenever the frequency is dynamically varied. In addition, the hardware to implement variable clock rate is also quite simple. We assumed that the frequency can be increased by 50%, 100%, or 300%, corresponding to C_r values of 0.75, 0.5, and 0.25.

4.1 Comparison Metric

We need to introduce a measurement index to determine the "optimal" point of operation. Since, the processor is going to make errors, traditional approaches such as delay, energy, or energy-delay product would be insufficient. We define the metric *energy-delay-fallibility* product, which is the product of the energy consumption, the execution cycles of the application, and the "*fallibility*" factor of the processor. The energy consumption is the energy consumed in the whole processor during the execution of the application. The *fallibility* is defined as the probability of the processor making an error for the application. One can use the number of hardware faults that are not detected to measure the fallibility. However, due to the application-specific nature of our target architectures, we use application errors in the fallibility factor as discussed in Section 2. Particularly, fallibility corresponds to the fraction of packets that have any type of errors. We also pay special attention to the fatal errors. Since fatal errors prevent other packets to be processed[3], we calculate the number of packets successfully processed till the occurrence of a fatal error. The reported energy-delay-fallibility factors are based on this number. We also report

[2] If the cache has sub-blocks, only the corresponding portions of the cache block can be invalidated and accessed from the level 2 cache. However, in this paper we do not study such cache structures.

[3] Majority the fatal errors we have observed during our simulations are because the execution gets stuck in an infinite loop. For such an error, the processor can be modified such that we can recover from the error.

the probability of a fatal error in addition to the energy-delay-fallibility product. Particularly, we record the probability of a fatal error with increasing clock frequency. Increased clock frequency makes system more susceptible to termination. As a result less number of packets can be processed successfully at higher clock frequency.

Although we argue that the packet processors can have faults, frequent faults are certainly undesirable considering the system behavior. Therefore, instead of giving the same weight to each component in energy-delay-fallibility product, one can give more weight to the fallibility. Particularly, the product can be calculated as energy^k-delay^m-fallibility^n according to the needs of the architecture. In our studies, since delay and fallibility are more important than energy, we set k to 1, m to 2, and n to 2. The energy-delay-fallibility product can be defined for a single component (e.g., cache). However, in this paper, we measure the metric for the applications.

5. Experimental results

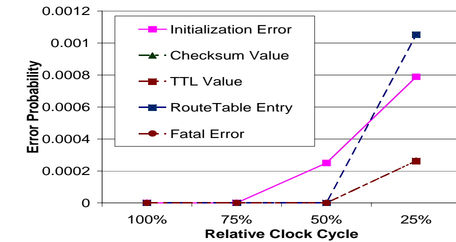
5.1 Simulation Environment

We use the SimpleScalar/ARM [7] for our simulations. We modified the processor configuration to model a processor similar to execution cores in a variety of Network Processor architectures. Particularly, we simulate a processor similar to StrongARM 110 with 4 KB, direct-mapped L1 data and instruction caches with 32-byte line-size, and a 128 KB, 4-way set-associative unified L2 cache with a 128-byte line-size. The level 1 data cache has 2-cycle latency and the level 2 cache latency is 15 cycles. We first modified the applications to mark the values of data structures mentioned in the previous section. Then, we have modified the simulator to introduce random faults into the execution and to simulate the effects of the introduced faults. We chose an initial fault probability of $2.59*10^{-7}$ per bit (in accordance with the formula (4)). This fault rate is similar to the rates reported by Shivakumar et al. [23]. The probability of a two-bit fault is set to $2.59*10^{-9}$, and the probability of three-bit faults is $2.59*10^{-10}$ in accordance with reported correlation between single-bit and multiple bit faults [12]. For the higher clock rates, we increase the fault rate in steps according to formula (4).

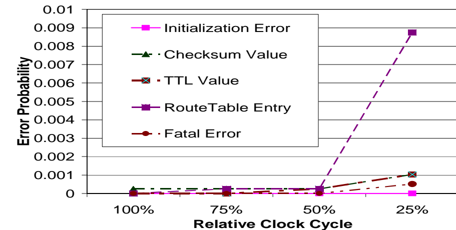
5.2 Application Error Behavior

This section describes the simulation results observed for the networking applications. The experiments in this section measure the effect of different fault rates on the data structures discussed in Section 2.

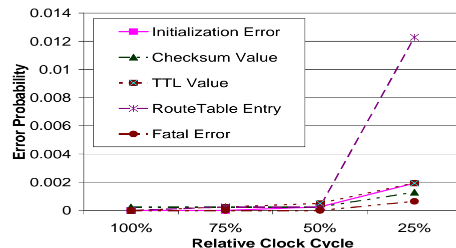
Figure 6 presents the results for the route application. For the results presented in Figure 6(a), we only introduce faults during the control plane tasks. Similarly, for the results in Figure 6(b), faults are introduced only during data plane tasks. For the results in Figure 6(c), faults are introduced during both the control plane and data plane tasks. Intuitively, the faults in the control plane tasks should have significantly more effect on the application behavior. This can be observed for initialization error when Figure 6(a) and Figure 6(b) are compared. However, for most error types, the difference is not drastic. This behavior is due to the shorter length of the control plane tasks compared to that of the data plane tasks. Therefore, although each fault happening during the control plane tasks has larger impact on the error rate compared to the faults during data plane tasks, the overall impact of varying the clock rate during the control plane tasks is not drastically more on the application errors. This is an encouraging result, because in many cases the processor will not have information about the type of task it is executing. Hence, it might be complicated to have different clock rates for different tasks. Since the results indicate that the effect of faults during control plane tasks is tolerable, we can "safely" vary the clock frequency.



(a) Faults introduced in control plane



(b) Faults introduced in data plane

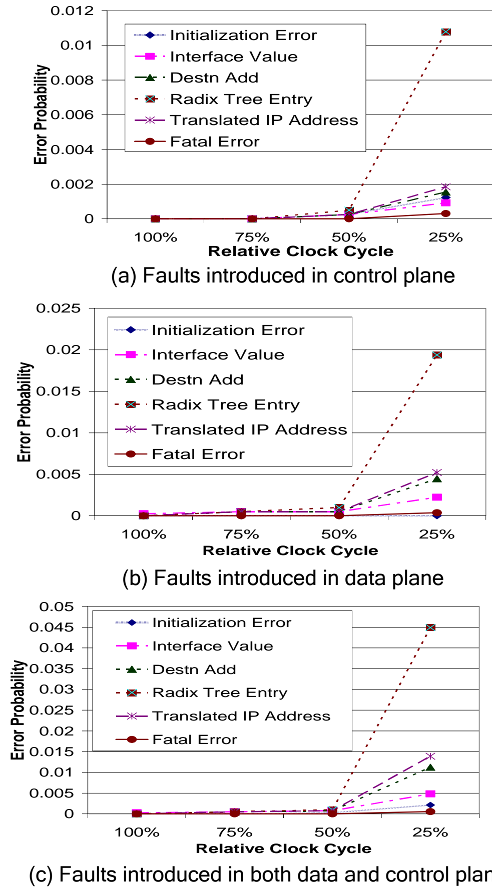


(c) Faults introduced in both data and control planes

Figure 6. Error Probability of route application

Figure 7 presents the results for the nat application. Similar trends can be observed for this application as well. Particularly for the nat application we see that errors due to faults during data plane tasks have more impact on the application behavior than the faults during control plane tasks. The results for the remainder of the applications are not presented due to their similarities with the presented results. However, all of them show identical characteristics of the applications under erroneous execution. Overall, all the applications can sustain faults to varying extents. For smaller fault rates we observed the execution of the application without any observable error in the data

structures and the application output. For larger fault rates, on the other hand, we encountered fatal errors and errors in the data structure values.



(a) Faults introduced in control plane

(b) Faults introduced in data plane

(c) Faults introduced in both data and control planes

Figure 7. Error Probability of nat application

With increase in the fault rates, the applications start producing erroneous outputs. Nevertheless, we see that the clock rate of the data cache can be increased up to 4 times without causing a major impact on the application output. Particularly, the largest error behavior we have observed in our simulations for the md5 application, when the relative clock frequency is set to 400%. For this particular case, the fallibility factor is 1.261. The fallibility factor of all the studied applications for the 0.5 and 0.25 relative clock rates are presented in the right-most columns of Table 1. The reasons we can increase the clock frequency as much as 4 times are two-fold. First, since the clock frequency is initially set too high by the circuit designer to be safe, increasing the clock frequency initially does not have a major impact on the fault probability. In addition, during the simulations, we have seen that not all the faults have an impact on the application output. On average we have only observed an error for approximately 15% of the faults.

5.3 Fatal Error Probability Measurements

We recorded the probability of a fatal error with increased clock frequency. Unlike other errors, fatal errors may destroy the system integrity. This prompts to ensure that the clock frequency should not reach a value that may result a high probability of fatal error. Figure 8 depicts the fatal error probability for different applications when there are no error detection schemes employed. Similar to the fallibility results, we see that the fatal error probability is zero for smaller increases in the clock rate. As we exceed 100% increase in the clock rate, we start seeing an impact on the fatal error probability.

Note that the fatal error probabilities in Figure 8 are measured for the base architecture, which does not employ any error detection scheme. Error detection schemes reduce the probability of fatal errors dramatically. In fact, during the simulations of the architectures with error detection, we have never encountered a fatal error.

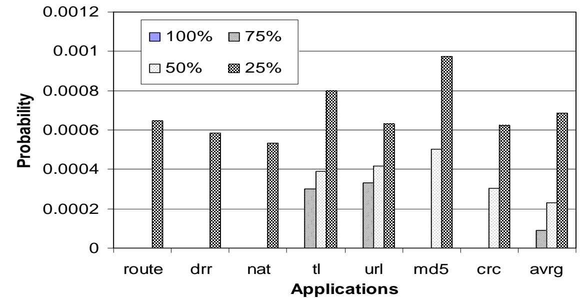
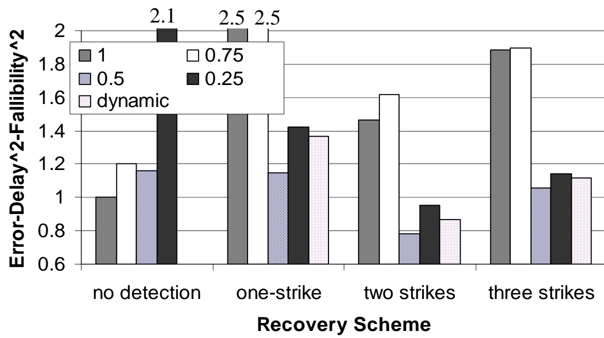


Figure 8. Fatal error probabilities for different clock rates.

5.4 Energy-Delay-Fallibility Measurements

The simulations presented in this section introduce faults during both the control plane and the data plane. As we have discussed in Section 4.1, different techniques are compared using the energy-delay^2-fallibility^2 product. To measure the energy consumed during the applications we use three models. For the energy consumption of the overall processor, we used the results presented by Montanaro et al. [14]. The energy consumed by the caches when they are operated with full frequency is found using CACTI [28]. When the clock frequency is increased, the voltage swing decreases. The energy consumed by the cache linearly shrinks with this decrease in the voltage swing. Therefore, we used the model presented in Figure 1(b) to find the relative voltage swing for different clock rates. Particularly, the energy consumed by the cache reduces by 45%, 19%, and 6% for relative clock rates of 0.25, 0.5, and 0.75, respectively. To estimate the energy consumed by the error detection scheme, we use the results presented by Phelan [17]. The level-1 data cache consumes 16% of the overall chip energy. Parity increases the energy consumed during reads by 23%. Similarly, the energy consumed during writes increases by 36%. We assumed that each word (32-bits) is protected by a single parity bit. To measure the delay in the applications, we calculate the average number of cycles spend for each packet. Note that we cannot use the total number of execution cycles, because some simulations do not finish to completion due to fatal errors. The fallibility factor is calculated as explained in Section 4.1.



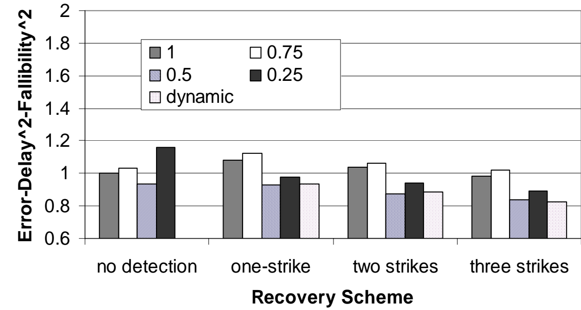
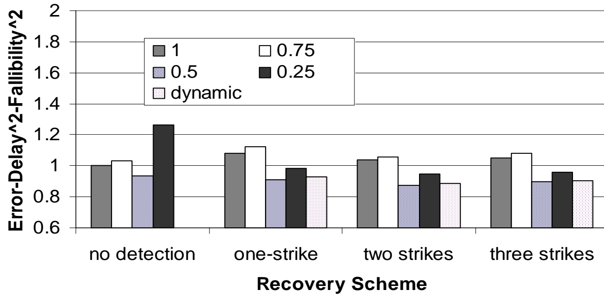


Figure 9: Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25: (a) For the route, (b) for the crc application. The bars represent the relative energy-delay^2-fallibility^2 product with respect to C_r = 1 with no-detection.



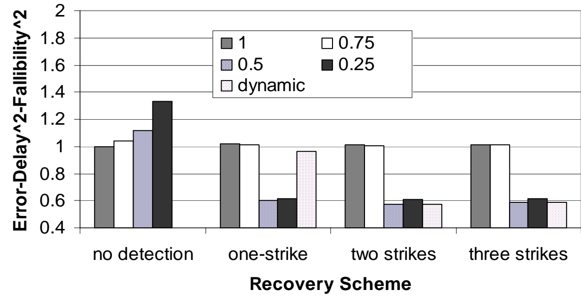
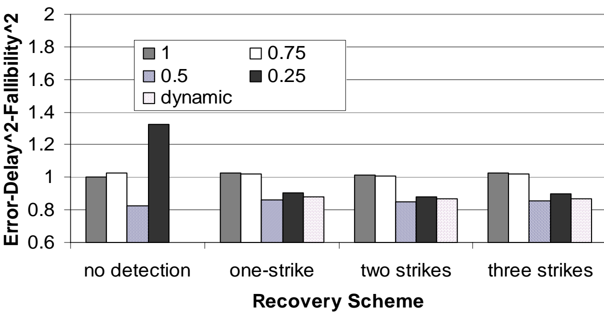


Figure 10: Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25: (a) For the md5, (b) for the tl application. The bars represent the relative energy-delay^2-fallibility^2 product with respect to C_r = 1 with no-detection.



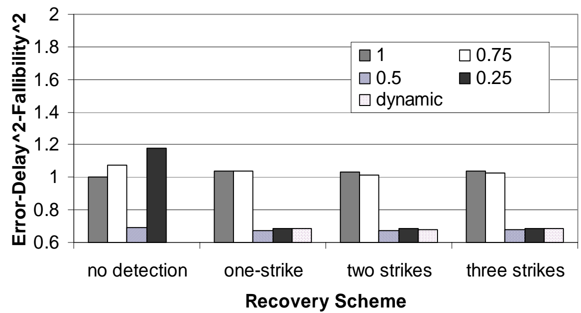
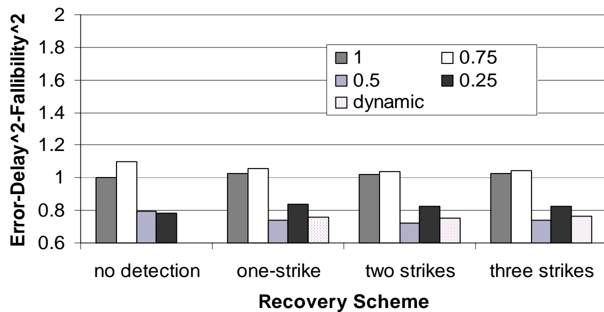


Figure 11: Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25: (a) For the drr, (b) for the nat application. The bars represent the relative energy-delay^2-fallibility^2 product with respect to C_r = 1 with no-detection.



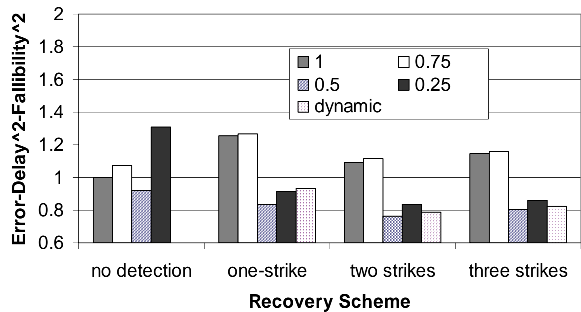


Figure 12: Energy-delay^2-fallibility^2 product for the simulated configurations, the dynamic configuration, and the static configurations with C_r = 1, 0.75, 0.5, and 0.25: (a) For the url, (b) for the average application. The bars represent the relative energy-delay^2-fallibility^2 product with respect to C_r = 1 with no-detection.

Results for the route application are summarized in Figure 9(a). For route application, we see that the best technique is the static technique with 50% relative clock cycle when two-strike recovery is used. For the crc application (Figure 9(b)), on the other hand, the best configuration is the dynamic frequency adaptation with three-strike recovery. When we compare these two applications, we see that crc is more resilient to faults, because due to its streaming nature it already has a large cache miss rate. Therefore, additional cache accesses due to errors have less effect on the execution time. As explained in Section 4, three-strike eliminates some of the incorrect accesses to the level 2 cache that might happen by the two-strike scheme. Therefore, three-strike improves the performance for the crc application because it reduces the pressure on the level 2 cache.

Figure 10(a) and (b) present the results for the md5 and tl applications, respectively. We see that similar to the route application, the static technique with 50% relative clock cycle and two-strike recovery scheme gives the best result. For the tl application, we see that the energy-delay^2-fallibility^2 product is reduced by as much as 43%. Tl application has a large fraction of load instructions. Therefore, reducing the cache access latency has a significant impact on the overall performance.

One interesting result with the tl application (Figure 10(b)) is the inability of the dynamic scheme to reduce the energy-delay^2-fallibility^2 product for the one-strike scheme. The reason for this is due to some initial errors, the dynamic scheme gets late into the 0.5 region. Since the total number of instructions executed for this application is small, the overall energy-delay^2-fallibility^2 remains high. The results presented in Figure 11(a) and (b) are for the drr and the nat applications.

Figure 12(a) presents the results for the url application. Figure 12(b), on the other hand, gives the average of all the simulated applications. Overall, we see that the static technique with 50% relative clock cycle and two-strike recovery scheme gives the best result reducing the energy-delay^2-fallibility^2 product by 24%. This is partially an artifact of the steps we have selected for the clock frequency. Although when we set C_r to 0.25, we see a significant reduction in the energy consumption, we also see a sharp increase in the error rates. Therefore, $C_r = 0.5$ almost always performs better than the $C_r = 0.25$. As a result, the dynamic scheme also stays mostly in the $C_r = 0.5$ region and hence does not perform better than the static scheme. Note that if we do not consider the errors, the static approach with $C_r = 0.5$ and two-strike recovery scheme reduces the energy-delay product of the processor by 17%, and the energy-delay^2 product by 26%.

In almost all the applications, we see that without the error detection, increasing the clock frequency increases the energy-delay^2-fallibility^2. The reasons for this are two-fold. First, we take the square of the fallibility in our metric. Since we increase the fallibility factor when we increase the clock frequency, there is a significant increase in our metric. Second, we see that errors usually increase the number of execution cycles. There are two reasons for this. First, erroneous load operations usually result in misses in the cache. More importantly, we see that the number of instructions executed also increases with the errors. This is mostly due to the loops. If one of the values that affect the completion criteria changes, we see that in most cases the number of iterations increase.

6. Related work

One class of related work is in the area of fault tolerance. Traditionally, fault tolerance has caught attention in the context of environments with heavy concentration of alpha-particles and atmospheric neutrons [27]. Transient faults induced by these particles are shown to decrease the reliability of processors [25]. Another area where there has been a strong emphasis on reliability is circuit verification, which is an important problem in IC fabrication. Techniques exist to study potential errors in the pre-silicon [5] stage and also subsequent to the fabrication process [1]. More recently, designing computer systems for resiliency [2] to transient faults has gained greater significance due to the combined effect of higher integration densities, lower voltages, and faster clock frequencies. There have been various studies utilizing redundancy to increase robustness for SMT processors [15, 20], for superscalar processors [19], and for CMPs [8]. All of these techniques aim to increase robustness. Our approach, on the other hand, reduces it. Although this might seem controversial at a glance, our motivations are similar to these studies: correctness cannot be achieved by optimizations only at the circuit level. However, we propose to deal with the errors at the higher levels instead of trying to eliminate them.

Validation methods such as fault injection are particularly attractive for estimating the dependability of computer systems [10]. Mukherjee et al. introduces the architectural vulnerability factor (AVF) for various processor components [16]. However, we are not aware of any study that investigates the application-level behavior of networking programs under hardware faults. More importantly, these studies still do not allow an incorrect execution of the program as we propose in this paper. Austin introduces DIVA, which is a method for enforcing correctness in processors which can make mistakes because of the lack of complete verification [3]. DIVA still aims to achieve correctness, whereas in this paper we reduce the probability of correct execution.

7. Conclusions

In this paper, we proposed the design and utilization of clumsy packet processors. Clumsy packet processors use

the robustness available in the networking applications to increase the efficiency of hardware structures while increasing their fault probabilities. Overall, this results in better execution efficiency and reduced energy consumption. Particularly, we have shown how the access delay and energy consumption of a data cache can be reduced while increasing the hardware faults during accesses. We developed a realistic model that estimates the fault probability of the cache for a given clock frequency. Thus, a clumsy processor can increase the clock frequency of its data cache and reduce its energy consumption. We have also defined various application-specific error metrics that is used to measure the "*fallibility*" of the processor. Particularly, we have proposed the energy-delay-fallibility product metric, which can be used to measure the trade off between the energy, execution time, and the error probability. We have presented a scheme to adapt the frequency of the data cache to adjust to the application requirements. Our simulations reveal that there is a significant gap between the specifications of the circuit designer and the optimal clock frequency in terms of energy-delay^2-fallibility^2 product. The technique that doubles the clock frequency while utilizing a parity-based error detection scheme and a two-strike recovery mechanism gave the best result on average, which resulted in 24% reduction in the energy-delay^2-fallibility^2 product.

8. Acknowledgement

We thank S. O. Memik and B. Mangione-Smith for providing valuable feedback on this paper. We also thank Y. Ismail and M. Chowdhury for their help in developing the error models. Finally, we like to thank M. C. Wildrick and S. Jevtic for invaluable input to this work.

9. References

1. Anghel, L.a.M.N. *Cost Reduction and Evaluation of a Temporary Faults Detecting Technique*. in *Design Automation and Test in Europe (DATE)*. March 2000.
2. Annavaram, M., J.M. Patel, and E.S. Davidson. *Data prefetching by dependence graph precomputation*. in *28th Annual International Symposium on Computer Architecture*. 2001. Göteborg, Sweden.
3. Austin, T. *DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design*. in *International Symposium on Microarchitecture*. Nov. 1999.
4. Baker, F., *Requirements for IP version 4 routers.* 1812., June 1995.
5. Bose, P. *Ensuring dependable processor performance: an experience report on pre-silicon performance validation*. in *International Conference on Dependable Systems and Networks (DSN)*. July 2000.
6. Braun, F., J. Lockwood, and M. Waldvogel. *Reconfigurable router modules using network protocol wrappers*. in *International Conference on Field-Programmable Logic and Applications*. Aug. 2001. Belfast / N. Ireland.
7. Burger, D. and T. Austin, *SimpleScalar Tool Set, Version 2.0.* June 1997, University of Wisconsin.
8. Gomaa, M. ,et al. *Transient-Fault Recovery for Chip Multiprocessors*. in *International Symposium on Computer Architecture*. June 2003. San Diego, CA.
9. HP, *Nonstop computing, http://nonstop.compaq.com.*
10. Iyer, R.K. *Experimental Evaluation*. in *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*. 1995. Pasadena, CA.
11. Krishna, C.M. and L.-H. Lee. *Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time Systems*. in *Real Time Technology and Applications Symp.* May 2000.
12. Li, L. *Soft error and energy consumption interactions: a data cache perspective*. in *ACM/IEEE International Symposium on Low Power Electronics and Design*. 2004.
13. Memik, G., W.H. Mangione-Smith, and W. Hu. *NetBench: A Benchmarking Suite for Network Processors*. in *International Conference on Computer-Aided Design (ICCAD)*. Nov. 2001. San Jose / CA.
14. Montanaro J., et al., *A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor.* IEEE Journal of Solid-State Circuits, 1996. **31**(11): p. 1703-14.
15. 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.
16. Mukherjee, S.S., C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. *A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor*. in *International Symposium on Microarchitecture*. Dec. 2003.
17. Phelan, R., *Addressing Soft Errors in ARM Core-based SoC*. Dec. 2003, ARM Ltd.
18. Project, T.F., *FreeBSD Operating System*.
19. Rashid, F., K. K. Saluja, and P. Ramanathan. *Fault tolerance through re-execution in multiscalar architecture*. in *International Conference on Dependable Systems and Networks (DSN)*. June 2000.
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. Rivest, R., *The MD5 Message-Digest Algorithm.* Apr. 1992.
22. Security, I.R.D., *RSA Security Downloads.*
23. Shivakumar, P., et al. *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*. *(DSN)*. June 2002.
24. Shreedhar, G.V. *Efficient Fair Queuing using Deficit Round Robin*. in *SIGCOMM'95*. Aug/Sep 1995. Camridge / MA.
25. Srinivasan, G.R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview.* IBM Journal of Research and Development, Jan. 1996. **40**(1): p. p. 77-89.
26. Srinivasan, J.R., *Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview.* IBM Journal of Research and Development, Jan. 1996. **40**(1): p. p. 77-89.
27. 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.
28. Wilton, S. and N. Jouppi, *An enhanced access and cycle time model for on-chip caches*. July 1995, Digital Western Research Laboratory, 93/5.