Leveraging Transactional Memory for Energy-efficient Computing below Safe Operation Margins

Adrian Cristal
Osman Unsal
Gulay Yalcin
Barcelona Computing Center, Spain
first.last@bsc.es

Christof Fetzer
Jons-Tobias Wamhoff
Dresden University of Technology, Germany
first.last@tu-dresden.de

Pascal Felber
Derin Harmanci
Anita Sobe
University of Neuchatel, Switzerland
first.last@unine.ch

Abstract

The power envelope has become a major issue for the design of computer systems. One way of reducing energy consumption is to downscale the voltage of microprocessors. However, this does not come without costs. By decreasing the voltage, the likelihood of failures increases drastically and without mechanisms for reliability, the systems would not operate anymore. For reliability we need (1) error detection and (2) error recovery mechanisms. We provide in this paper a first study investigating the combination of different error detection mechanisms with transactional memory, with the objective to improve energy efficiency. We notably introduce an analytical model that allows us to give recommendations for future work.

1. Introduction

The increasing power and energy consumption of modern computing devices is perhaps the largest threat to technology minimization and associated gains in performance and productivity. For instance, current scaling trends have led to multi-core processors at the architectural level, and higher core counts are expected in the following years. Yet, it will not be possible to keep all the cores on the whole chip powered-on at the same time due to power envelope issues, a problem also known as the dark silicon phenomenon [13].

As the power envelope becomes one of the key design concerns for computer systems, a dramatic improvement in the energy efficiency of microprocessors is required in order to keep the power under control. Since energy consumption grows proportionally to the square of supply voltage (i.e., $Energy \approx C_L \times V_{dd}^2$), a very effective approach in reducing the energy consumption is to reduce the supply voltage (V_{dd}) close to the transistors' threshold (near-threshold execution) or lower than the threshold (sub-threshold execution). Voltage downscaling can offer substantial energy savings by trading off performance. To take advantage of potential power savings, microprocessors have started to provide high-performance and low-power operating modes [20]. While the processor runs at a high frequency by using high V_{dd} in the high-performance mode, in the low-power mode the processor reduces V_{dd} and the frequency.

However, the energy reduction in the low-power mode comes with a drastic increase in the number of failures [9]. As V_{dd} decreases, failure rates for yield loss, hard errors, erratic bits, and soft errors increase.

In order to fully exploit the dynamic energy savings of voltage downscaling, a potentially attractive idea is to implement reliability solutions that allow a system to operate below the safe margin of V_{dd}. In this position paper we investigate the usefulness of the combination of two key capabilities: (1) error detection and (2) error recovery. Error detection is the process of discovering that an error has occurred, while error recovery is the process of restoring the system's integrity after the occurrence of an error.

The combination of error detection and recovery for dependable multiprocessor systems is not new, implementations of embedded systems as well as supercomputers often rely on checkpointing and rollback that are triggered on error detection [42].The quality of the error recovery for energy efficiency is, however, even more critical. A poorly implemented checkpoint/rollback mechanism might consume more energy than saved by reducing the voltage of the processor. We are therefore interested in new, lightweight mechanisms. One of the possible solutions is to use transactional memory (TM), which provides automated checkpointing/rollback. TM was originally introduced to simplify the process of parallel programming [18], but is also used for implementing reliability as shown in [14]. We believe that TM can simplify the process of energy-efficient reliable programming in a similar way. Researchers showed that the use of TM can consume less energy than traditional lock-based mechanisms, for micro benchmarks [25] and also for the more sophisticated STAMP benchmarks [16]. Hence, it is worthwhile to further investigate TM in combination with different error detection mechanisms for processors working at low voltage levels.

In this paper, we survey different existing error detection mechanisms (Section 2) and TM solutions (Section 3) that apply for error recovery. In Section 4 we provide an initial study by an analytical model and show that it is possible to combine error detection and TM if sufficient hardware support is provided. We further investigate the edge cases on voltage reduction while the error recovery still leads to a reduced energy consumption. Finally we give an outlook and conclude the paper.

2. Error Detection Schemes

As V_{dd} decreases, the bit failure rate increases rapidly [6, 24]. The nature of these failures due to voltage level reduction are either transient (i.e., soft errors and erratic bits) or permanent (yield loss and hard errors). Permanent faults can be detected by online detection schemes applied at boot time, or at the change of the supply voltage [7]. Transient failures are dynamic and therefore are not easy to detect. Moreover, for recovery re-execution is needed. In this work, we focus on transient failures and consider different detection and recovery schemes for dealing with such failures.

We review several lightweight detection mechanisms and discuss their applicability for energy efficient computing. Typical error detection mechanisms found in the literature (1) run the code redundantly and compare the outputs, i.e., rely on replication, (2) use assertions/invariants, (3) use encoded processing, (4) use approximate computing, or (5) monitor the error symptoms.

We further discuss how these error detection mechanisms can be combined with TM (irrespective of whether TM is built in software or hardware). A qualitative comparison is provided at the end of the section.

2.1 Replication

To satisfy the strict reliability requirements of mission-critical systems, various redundancy-based error detection solutions have been proposed [31, 36]. Triple modular redundancy (TMR) schemes execute the instruction stream three times, expecting a single result. A voting circuit decides, upon result divergence, which replica is correct, thus no error recovery is needed. Dual modular redundancy (DMR) schemes execute an instruction stream redundantly in two synchronized processors and check if both produce identical results. If the results diverge, a recovery mechanism can be triggered. The comparison of execution results causes synchronization and comparison overheads in execution time. Another issue is the non-determinism introduced by thread scheduling or user input/output, which makes the synchronization of the replicas more complex and invasive.

Using TM for dual replicated execution reduces the comparison overhead [38, 39]. This is because, instead of comparing each individual store, one can efficiently compare the write-sets (which typically have less entries than the total number of store instructions because multiple stores to the same address are mapped to a single entry). Also the comparison is done only at the commit stage of the transactions, which provides implicit checkpoints. The comparison overhead can be further reduced by comparing hash-based signatures of write-sets and register files of the transactions.

Although replication provides a very high error detection capability, it suffers from 100% energy and space overhead in the error-free execution.

2.2 Assertions/Invariants

Assertions are a common technique for detecting software or hardware errors [3]. Assertions are conditions referring to the current and previous state of the program. If the states do not match the expected results, an error is detected. Upon such event, the typical behavior is to issue a warning, but corrective actions can also be triggered [27].

An approach based on a coprocessor (watchdog) is proposed in [23]. It uses annotations in the first phase of the error detection, where processes provide some information. In the second phase the processes are continuously monitored and the collected information is compared with the information previously provided. The authors claim an error coverage of 90% of transient and permanent errors by control-flow and memory access checking.

As pointed out in [14], combining assertions with transactions is an interesting approach as one can implicitly create the latter based on the invariants provided by developers. Inserting invariants manually into the program has the drawback that the resulting assertions might be unsound (lead to false positives) or incomplete [21] and might be inefficient because too many evaluations are needed. The alternative is to add them automatically to a program, as proposed in [12].

The authors of [17] propose an extension of STM Haskell with invariants that concentrates on C like consistency from the ACID characteristics. Consistency is ensured by dynamically-checked invariants that must hold if the system is in a consistent state. The authors identified that the frequency of invariant evaluation represents a tradeoff between overhead and detection rate. In their work they reduce the overhead by the following measures. The invariants are (1) garbage-collected if their watched data structure does not exist anymore, and (2) invariants are only checked if a transaction wrote a variable read by the invariant.

2.3 Encoded Processing

Error correcting codes (ECCs) are commonly used to detect and correct soft errors in memory by adding redundancy. ECCs usually provide single bit error correction and double bit error detection [41]. However, soft errors might also be introduced during data transport and processing in the logic building blocks. One way of applying the principles of ECC to runtime errors is encoded processing [15]. The redundancy is added by applying arithmetic codes to the values processed by the application. This can be done either using custom hardware or in software by an encoding com-

piler [35]. All operations must preserve the encoding, which results in more computations and higher energy consumption.

The level of error detection that can be achieved using encoded processing depends on the selected type of arithmetic code, e.g., AN codes can detect value errors while ANBD-mem codes [30] can additionally detect lost updates in memory, but at the expense of a higher processing overhead [29]. The observed rate of undetected errors is 9% and 0.5%, respectively.

If we combine encoded processing with transactional memory, a value is validated when it is read or written by checking its arithmetic code. If the code is incorrect, the transaction must be aborted. For higher efficiency, the validation of a code word can be deferred until a transaction commits or the value becomes externally visible (lazy checking). This avoids the costly check on each access of the value because the error propagates in the employed arithmetic code. Eager checks can allow the application to identify the first occurrence of an invalid value and to react more proactively.

2.4 Symptom-Based Error Detection

In order to provide reliability at a low cost, some recent error detection solutions [22, 34] monitor program executions to inspect if there is a symptom of hardware faults. These symptoms can be mispredictions in high confidence branches, high OS activity, or fatal traps (attempting to execute an undefined instruction code).

Recently, symptom-based error detection mechanisms using transactions to recover from application crashes have been proposed in SymptomTM [40] and disclosed in a patent filed by IBM [5]. The system starts a single special transaction at the beginning of the application. The execution of the transaction is monitored to detect if there are any symptoms of hardware errors, which typically result in fatal traps (e.g., undefined opcode). Unless any fatal trap exception is raised in the transaction, the write-set is committed to shared memory at the end of the transaction. Otherwise, the system aborts and re-executes the transaction. Since there is no replication, the scheme has virtually no area/energy overheads. It has, however, limited error coverage since it cannot detect data corruptions and, further, exceptions can be raised after the commit of the transaction.

Since some symptoms can be observed very efficiently (e.g., catching exceptions), symptom-based error detection can be easily combined with other error detection mechanisms. Other symptoms (e.g., infinite loops due to a corruption of the stop condition) require an instrumentation of the code or support by the operating system (e.g, adding timeouts to operations).

2.5 Approximate Computing

An alternative to encoded processing and redundant execution is approximate processing. Identifying non-critical instructions and executing them using approximate processing by exploiting narrow values for integer code is becoming common practice [33]. A further measure is to tolerate errors when they occur in non-critical instructions or in the non-significant bits when absolute precision is not required (floating point code). This model can be considered as a best-effort approach, such as seen in network and storage systems [4]. The authors claim that some of the guarantees (like precision) should be shifted from the processing unit to the application in order to gain performance and scalability, and to support execution on unreliable hardware. The authors propose to split applications in critical and non-critical (best-effort) computations.

An error occurring in the computationally non-critical bits can be safely ignored, thereby reducing the energy overhead of error detection. The task of identifying the non-critical parts of the code might be transferred to the programmer by the use of annotations. The critical computations should be monitored by a combination of error detection and recovery. We believe in particular that a combination of error detection and TM with approximate computing is worth future investigations.

Method	Memory	Processing
Replication	high	high
Assertions	medium	high
Encoded Processing	medium	high
Symptoms	low	low

Table 1. Memory and Processing Overhead Comparison

2.6 Qualitative Comparison

Error detection is a critical step for enabling low voltage operation but it does not come without cost. The energy efficiency is highly dependent on the selection of the right technologies. In the following, we summarize the aforementioned schemes and provide a comparison regarding factors that influence the design decision. We concentrate on (1) the overhead introduced in memory and processing, (2) the error detection coverage, (3) the requirements for setting up the error detection.

Memory and Processing Overhead. Table 1 compares the overhead of the single error-detection schemes in processing and in memory, when applied to an error-free system.

Replication has high memory and processing overheads because the whole application executes in parallel. With assertions/invariants, the overhead depends on the programmer or the automated tool that generates them. It can be medium to high in memory, depending notably on the support for garbage collection. The annotations have to be evaluated in any case (even if there are no failures). Encoded processing needs only a small amount of additional memory to keep the arithmetic codes, but all executed operations incur the significant overhead of maintaining the encoding. For the

symptom-based error detection only the symptoms have to be stored and checked, therefore the overheads are low.

Error Detection Coverage. There is usually a tradeoff between error detection coverage and overhead. For example, whereas replication provides 100% error detection, it requires many resources and hence might not be usable for energy efficient computing when the processor runs in high performance mode. The assertion-based mechanism is highly dependent on the implementation. Transient errors might not be detected, because they are simply not covered. However, there are implementations that claim to reach 97% coverage with only 5-14% performance overhead [28]. The detection capabilities of encoded processing depend on the applied arithmetic code. Its complexity introduces linearly increasing runtime costs, while the error detection rate increases exponentially [29]. Symptom-based error detection provides a limited error coverage with a very-low performance overhead.

Requirements for Using the Mechanism. Although the energy efficiency of a system is the main goal, a mechanism can only be successful if it can be easily applied, especially if the error-detection mechanisms have to be combined with recovery. Replication has few requirements for the checking of the output and the application does not have to be changed. Assertions usually need language support to be defined and the implementation must verify them during the execution. Encoded processing can be implemented as a combination of compiler and library, and integration with the application is straightforward.

3. Error Recovery

Error detection is not sufficient by itself to ensure the reliability of a system; it needs to be coupled with an error recovery solution. There are mainly two categories of recovery mechanisms: forward and backward error recovery (FER and BER). FER is based on replicating the execution in order to use the correct results if the actual execution fails. This approach assumes that the replicated execution is error-free and, hence, has limited interest when operating at low voltages (one example is TMR). BER (also called checkpoint/rollback mechanism) stores an error-free state of the system (checkpoint) and reverts the system state upon error detection (rollback). BER is classified in three groups according to the checkpointing strategy used.

Global Checkpointing [19, 26, 32]. Periodically, all processors are synchronized to store a global checkpoint. The scalability of this approach is limited as it introduces the significant overheads: (1) during barrier synchronization performed at checkpointing some processors might stay idle if load is not properly balanced between them (e.g., some processors perform I/O operations before the checkpoint), (2) the recovery requires all processors to rollback to an earlier validated state, which causes unnecessary rollbacks of the error-free processors.

Coordinated-local Checkpointing [2, 37]. The overheads of global checkpointing are mitigated by synchronizing only the set of processors that have communicated with each other between two checkpoints to decide on a common checkpoint, whereas all other processors can perform local checkpoints. This approach has been shown to outperform global checkpointing [1].

Uncoordinated-local Checkpointing [10, 11]. In contrast to the two previous approaches, uncoordinated-local checkpointing performs checkpointing locally at each processor without any synchronization and also stores the interactions between processors in order to rollback to a consistent checkpoint. This approach is interesting for executions where processors communicate rarely.

Checkpointing can also be supported by transactional approaches. In particular, the use of TM for handling transient faults has been been proposed in Yalcin et al. [38]. We discuss next how TM can be used to implement error recovery.

3.1 Adapting TM for Recovery

Although TM (and especially STM) is known to have a high overhead for certain workloads, a significant portion of this overhead is due to data synchronization when detecting whether different threads accessed common data. For error recovery purposes, however, only the checkpoint/rollback behavior is necessary and the synchronization requirement is therefore largely reduced. Hence, it is possible to design cost-effective TM for error recovery by providing minimal synchronization (e.g., [38]). Such TM designs can easily provide coordinated-local checkpointing.

The cost of providing checkpoint/rollback behavior depends mainly on the logging strategy. *Redo-logging* (lazy data versioning) performs speculative modifications on private copies and makes the modifications effective on shared memory only at transaction commit. Since the modifications within transactions are repeated—at least once for the private copy and once for the shared memory—a significant overhead is introduced even to error-free executions. *Undo-logging* (eager data versioning) performs in-place memory updates during transaction execution and introduces overhead only upon abort, i.e., upon error recovery. The abort overhead is caused by the replacement of modified versions of data with their versions prior to the transaction. As such, undo-logging is preferable in terms of performance and energy efficiency.

While having lower time overhead, undo-logging can easily result in the propagation of a fault between concurrently executing tasks, because speculative changes become effective immediately on shared memory locations. Therefore, a synchronization mechanism is needed for error recovery. Conversely, high fault rates can cause important overheads and it may be more interesting in such cases to use redo-logging. This would reduce the synchronization costs because fault propagation can only occur during transaction commit.

Using undo- and redo-logging simultaneously can increase the recovery reliability provided by the TM. The rationale is to use undo-logging for enabling rollback, while using redo-logging to build a (correct) history of updates done by the transaction until the error is detected. The introduced overhead is limited to an additional replication of write operations, while the operand is already in the cache. Since transaction rollback cannot be guaranteed to be error-free when operating at low voltage, the history can be used as FER during the re-execution of the transaction to mask any faults. Doudalis and Prvulovic [8] proposed combining undo- and redo-logging to support both bidirectional debugging and error recovery, which can be another research direction for TM. We limit however the scope of this study to the previously evaluated error recovery scheme and do not further discuss the usage of undo-logging.

3.2 Executing with TM for Recovery

In order to integrate TM within an error recovery scheme, the code that requires recovery should be executed within transactions regardless of whether the original code includes transactions or not. We name the process of executing a code within a transaction as *transactification*. The need for transactification raises two issues: (1) determining the scope of transactification, (2) choosing the right transaction granularity.

Within the context of transient faults, a TM should be capable of taking control of the executed code at any time, since the voltage level can be reduced at an arbitrary moment. This implies that the scope of transactification should span the entire code, except code that explicitly declares that transactification is not needed, e.g., non-critical sections in approximate computing. If STM is used, all code (not only applications but also operating system code) running on a machine should also have a transactional version to switch to transactional execution at any time. For a hardware TM (HTM), transactification is done transparently in hardware, but the size of a transaction is limited. If the size of transaction can be kept small, it is possible to use an HTM alone. Otherwise a hybrid TM is required, i.e., where the HTM limitations are exceeded, STM is used as a fall-back.

Determining where the transactions start and end during low voltage operation is also an important issue. Inserting the executed code inside a single transaction is not feasible, since this requires an unbounded buffer in order to store the unmodified states of all modified data (the transaction size cannot be known in advance). Hence, once a core starts operating at low voltage we need to execute the code inside back-to-back transactions with known write-set sizes. It is further important to take care not to miss errors if there is a delay between occurrence of a fault and its detection. Otherwise, an instruction might be already committed even though execution was faulty. It is possible to introduce delays to ensure that all the instructions within a transaction completed without errors. At this point, the choice of the transaction granularity is critical. Small transactions permit efficient TM implementations (e.g., HTM) but may introduce too many artificial delays, slowing down the error-free execution. Large transactions can hide the artificial delays, but make it difficult for the TM to be efficient (e.g., requiring STM at least as a fall-back).

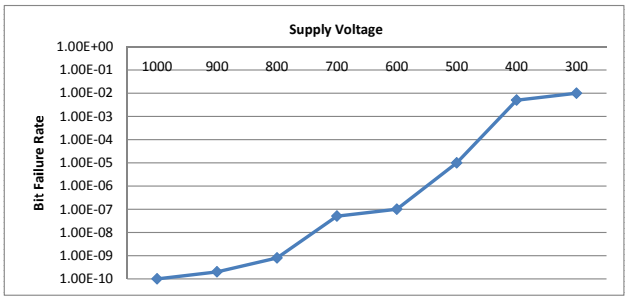

Figure 1. Bit failure rate of transient faults versus supply voltage V_{dd} (log scale) [6]. As V_{dd} is lowered the bit failure rate increases exponentially.

4. Analysis

In this section we analyze the feasibility of applying the aforementioned combinations of error detection with TM-based error recovery. We are specifically interested how much we can lower the voltage while still providing high reliability and only introducing low overheads. We start with an estimation of the overhead of the TM-based error recovery. Then, we analyze the error detection capabilities (i.e. provided reliability) under the given supply voltage. Finally, we analyze the possible energy minimization of combinations of error detection and recovery by considering the provided reliability of the schemes.

In order to approximate the energy spent for error recovery using redo-logs, we need to estimate the number of failing transactions. Note that we consider light-weight TMs that target reliability and are supported by hardware rather than regular TM with the purpose of concurrency control. Thus, they do not require code transformation and they can be committed when it is required. In order to estimate the number of failing transactions, we need three key parameters (1) the bit failure rate at a given supply voltage level (2) the size of the architectural structure and (3) the size of the transactions.

The relation between bit failure rate (i.e., the probability that a given bit fails) and supply voltage V_{dd} has previously been examined by Chishti et al [6] Here, we focus on bit failure rate caused by transient faults which we present in Fiure 1. The figure depicts that the decrease of V_{dd} results in an exponential increase of the bit failure rate. For instance, at V_{dd}=1,000 mV there are practically no failures (bit failure probability is 10^{-10}) while with V_{dd}=300 mV the bit failure probability increases to 1 %.

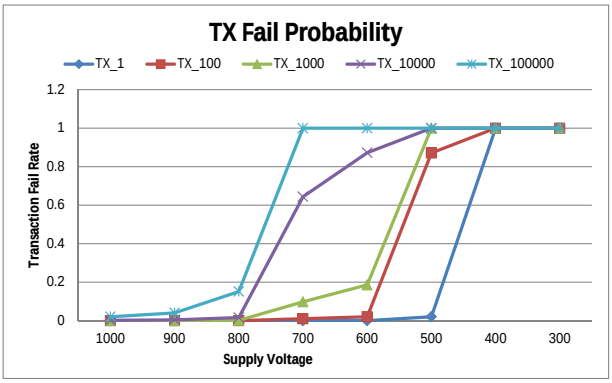

Figure 2. Transaction fail probability versus transaction size. As transaction size increases, the probability that the transaction fails increases as well.

In this initial study, the architectural structure only considers the register file, since the biggest SRAM structure in an in-order core is the register file, assuming that caches are protected by other means. Note that the current trend for multi-core processors is to put more, simple in-order cores instead of putting fewer, complex out-of-order cores. Also in-order cores do not require several buffers utilized in out-of-order cores such as issue queue, re-order buffer, etc. In this study, we evaluate a register file with 32 entries of 64-bit registers. We assume the worst case scenario in which any bit failure in the register file is harmful for the correctness of the execution. In the following equations we show the failure probability of the register file p_{RF}. Let $p_{coreBit}$ being the bit failure probability (presented in Figure 1).

$$p_{RF} = 1 - (p_{RFNoFault})$$

$$p_{RF} = 1 - (1 - p_{coreBit})^{RFSize}$$

For the third parameter, the transaction size, we do not make any assumption about the type of transactions (i.e., lazy/eager conflict detection or hardware/software transactions) since each error detection scheme determines it. Reliability purposed transactions can be committed when it is required, hence, the size of them can be defined based on the requirements. We calculate the probability that a transaction has a fault (p_{TXf}) with the following equation, where p_{TXr} describes the probability of a reliable transaction and s describes the size of a transaction.

$$p_{TXf} = 1 - p_{TXr}$$

$$p_{TXf} = 1 - (1 - p_{RF})^{s}$$

We use this equation to show that the energy consumption is highly influenced by the transaction size and present the results in Figure 2. Obviously, larger transactions have a higher failing probability when the supply voltage is below safe margins. For instance, when we use transactions with 100,000 instructions and the V_{dd} is lower than 800 mV, it is very likely that all transactions will observe at least one failure. This graph further shows that the minimum useful V_{dd} is 500 mV, whereas the transaction consists of only 1 instruction. This is, however, not realistic and therefore we consider transactions with 100 instructions for further analysis. This means that the lowest feasible voltage is 600 mV.

Limiting transaction sizes to 100 instructions is straightforward since reliability purposed transactions can be committed when it is required. As described by Yalcin et al [39], the size of reliability purposed transactions are less than 1,000 instructions when the write-set is limited to 64 entries. When TM is used for energy minimization, these transactions should be of finer granularity by using smaller write-sets.

The granularity of a transaction is one important point, another point is the reliability of the error detection mechanism. The application reliability (R) depends on the capability to detect failed transactions and on the number of transactions in the application. We model the reliability according to the following equations, where TX_R describes the reliable transactions, TX_F the failed transactions, a the number of transactions (application size) and $notDetected$ the rate of failures that are not detected.

$$R = (TX_R)^a$$

$$R = (1 - (TX_F * notDetected))^a$$

The capability of the error detection mechanisms differ. For instance, triple modular redundancy (TMR) cannot correct a fault if three copies of the execution are faulty. For double modular redundancy (DMR), if two register files in the coupled cores are erroneous in the same bit position, DMR can not detect the error. For symptom-based error detection, only 35 % of the failures can be detected when the transactions are short (i.e., 100 instructions) [40]. Similarly, encoded processing can detect 95 % of faulty transactions while invariants can detect 97 % of the failed transactions.

In Figure 3, we present the reliability of an application under the given V_{dd} when TM-based implementations of these error detection schemes are used. In this model, we assume that the application consists of 10,000 transactions. We also defer benign faults by assuming that any undetected fault leads to an incorrect result. As it can be seen in the figure, none of the schemes works when the V_{dd} is lower than 500 mV. TMR and DMR have a similar reliability and support voltages until 600 mV with 100 % reliability and 80 % until 500 mV. Invariant and encoded processing have a reliability of more than 90 % until the supply voltage of 800 mV. Symptom-based error detection cannot provide more than 90 % reliability, even in high performance mode.

Given these results, we evaluate the energy consumption overhead of the before mentioned mechanisms. In this simplistic model, we ignore the energy spent for comparing the write-sets. Also, we assume that the redundant executions

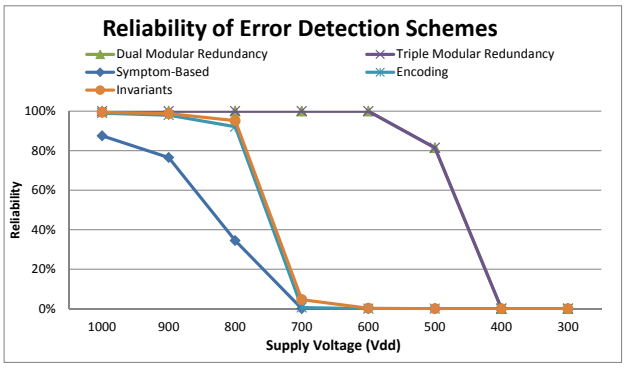

Figure 3. The reliability of the application according to supply voltage under several error detection schemes.

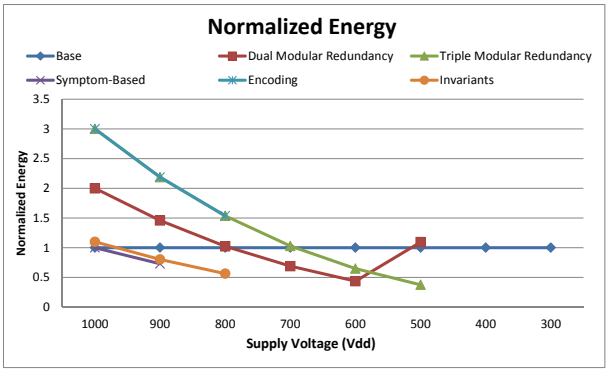

Figure 4. Energy reduction of each error detection scheme. Energy consumptions are normalized to the base case where the processor operates with 1,000 mV supply voltage.

are scheduled to different cores to detect the intermittent faults as well. One can argue that redundancy can be accomplished with less energy overhead by utilizing Simultaneous Multi-Threading but we did not explore this in our model. The typical energy consumption is approximated as the product of capacitance, voltage and frequency with the execution time (T):

$$E \approx C_L \times V^2 \times f \times T$$

With DMR the energy consumption doubles $E_{DMR} = 2 \times E$, with TMR it is triplicated $E_{TMR} = 3 \times E$. Encoded Processing has an overhead factor of 3 ($E_{EP} = 3 \times E$) since, for example, an encoded addition takes three times as long as an unencoded one [35]. Invariants present, on average, 10 % energy overhead $E_I = 1.1 \times E$. Symptom-Based detection does not present a perceptible error detection overhead $E_S = E$.

For error recovery, we assume that all evaluated mechanisms use the abort mechanism in a lazy data versioning transactional memory system, except TMR and invariants, which use forward recovery. The main overhead comes from the re-execution of faulty transactions. However, after aborting the transaction, another failure may occur in re-executing the transaction. Thus, assuming that the failing probability of a transaction is p_{TXf} and energy required for re-execution is E_{TX}; we calculate the energy spent for recovery ($E_{Recovery}$) with the following equations:

$$\begin{aligned} E_{Recovery} =& E_{TX} * p_{TXf} + E_{TX} * p_{TXf} * p_{TXf} \\ &+ E_{TX} * p_{TXf} * p_{TXf} * p_{TXf} + ... \end{aligned}$$

$$E_{Recovery} = E_{TX} * (p_{TXf} + (p_{TXf})^2 + (p_{TXf})^3 + ...)$$

$$E_{Recovery} = E_{TX} * p_{TXf} * (1/(1 - p_{TXf}))$$

In Figure 4, we calculate the energy consumption of each scheme for error detection and error recovery under the calculated failure rate of the given V_{dd}. We normalized each value to the base case, which executes in 1,000 mV supply voltage without any error detection and recovery. We omit the values where the schemes have zero reliability. At V_{dd}=1,000 mV one can see the error detection overhead, as the error rate is low.

Applications that require a reliability level close to 100 % (i.e., mission critical applications) should pay the error detection overhead by using either TMR, DMR, encoding or invariants and operate on high voltage levels. The symptom-based scheme can reduce the energy consumption by 27 % for V_{dd}=900 mV and provides 77 % reliability. Obviously, this scheme cannot be used for mission-critical applications. Similarly, invariants reduce the energy consumption by 44 % when V_{dd}=800 mV by providing 95 % application reliability. DMR reduces energy by 56 % when V_{dd}=600 mV and provides a reliability level close to 100 %. After 500 mV, DMR requires several rollbacks, thus, it increases the energy consumption drastically. TMR, on the other hand, does not execute rollbacks and the result is found by a majority voter. Thus, it can provide 63 % energy reduction by providing 80 % reliability when V_{dd}=500 mV.

Encoded processing cannot reduce the energy consumption as much as the other schemes due to its high error detection overhead presented by the software. We argue that hardware support for encoding can reduce this overhead drastically.

We showed that the combination of error detection and TM-based error recovery can be used when lowering voltage. The decision on which schemes should be selected is dependent on the required level of reliability (i.e. if applications are mission-critical or not) and the targeted supply voltage. A possible solution would be to provide all mechanisms and decide adaptively based on hints by the application (e.g., configuration), which of the mechanisms to use.

5. Conclusion

To improve the energy-efficiency of modern CPUs, one can reduce the supply voltage of cores. Reducing the supply voltage increases however the likelihood for wrong executions of programs. In this paper, we proposed to use transactional memory (TM) for rolling back the effects of wrong executions. To reduce the energy consumption, one needs an error detection scheme that has both a sufficient coverage and a low overhead. We discussed multiple error detection alternatives. Based on our evaluation, we conclude that one can reduce the energy consumption of CPUs—in particular, if we have efficient hardware support for TM and for error detection. An open question remains with respect to how effectively protect the TM itself against transient errors caused by low supply voltage.

Acknowledgments

This research has been funded in part by the European Community's Seventh Framework Programme [FP7/2007-2013] under the ParaDIME Project (www.paradime-project.eu), grant agreement no. 318693.

References

[1] R. Agarwal, P. Garg, and J. Torrellas. Rebound: scalable checkpointing for coherent shared memory. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 153–164, 2011.

[2] R. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (carer) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 82 – 88, jun 1990.

[3] D. Andrews. Using executable assertions for testing and fault tolerance. In *9th Fault-Tolerance Computing Symp*, pages 20–22, 1979.

[4] S. Chakradhar and A. Raghunathan. Best-effort computing: re-thinking parallel software and hardware. In *Proceedings of the 47th Design Automation Conference*, pages 865–870. ACM, 2010.

[5] D. Chen. Local Rollback for Fault-Tolerance in Parallel Computing systems, United States Patent Application, 12/696780. 2011.

[6] Z. Chishti, A. R. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu. Improving Cache Lifetime Reliability at Ultra-Low Voltages. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–99, 2009.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 97–108, 2007.

[8] I. Doudalis and M. Prvulovic. Euripus: A flexible unified hardware memory checkpointing accelerator for bidirectional-debugging and reliability. In *ISCA*, pages 261–272, 2012.

[9] R. G. Dreslinski et al. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[10] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[12] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27 (2):99–123, 2001.

[13] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[14] C. Fetzer and P. Felber. Transactional memory for dependable embedded systems. In *7th Workshop on Hot Topics in System Dependability (HotDep)*, pages 223–227. IEEE, 2011.

[15] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFAC/IFIP/IFORS Symposium*, pages 79–84, 1989.

[16] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of the USENIX conference on Power-Aware Computing and Systems*, pages 6–6. USENIX Association, 2012.

[17] T. Harris and S. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06), Ottowa*, 2006.

[18] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[19] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 289 –298, jun 1995.

[20] J. P. Kulkarni et al. A 160 mV Robust Schmitt Trigger Based Sub-threshold SRAM. *IEEE Journal of Solid-State Circuits*, 42(10):2303–2313, October 2007.

[21] N. Leveson, S. Cha, J. Knight, and T. Shimeall. The use of self checks and voting in software error detection: An empirical study. *Software Engineering, IEEE Transactions on*, 16(4): 432–443, 1990.

[22] M. Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS*, 2008.

[23] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, 37(2):160–174, 1988.

[24] T. N. Miller, R. Thomas, J. Dinan, B. Adcock, and R. Teodorescu. Parichute: Generalized Turbocode-Based Error Correction for Near-Threshold Caches. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchi-*

tecture, pages 351–362, 2010.

[25] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Proceedings of the international symposium on Low power electronics and design*, ISLPED '05, pages 331–334, 2005.

[26] Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 111–122, 2002.

[27] L. Pullum. *Software fault tolerance techniques and implementation*. Artech House Publishers, 2001.

[28] S. Sahoo, M. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, DSN. IEEE International Conference on*, pages 70–79. IEEE, 2008.

[29] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *The Third International Conference on Dependability*, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[30] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Computer Safety, Reliability, and Security*, volume 6351. Springer Berlin / Heidelberg, 2010.

[31] T. J. a. Slegel. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19:12–23, 1999.

[32] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, pages 123–134, 2002.

[33] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. Macaco: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer-Aided Design*, pages 667–673. IEEE Press, 2010.

[34] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *TDSC*, 3:188–201, 2006.

[35] U. Wappler and M. Müller. Software protection mechanisms for dependable systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 947–952. ACM, 2008.

[36] A. Wood, R. Jardine, and W. Bartlett. Data integrity in HP NonStop servers. In *Workshop on SELSE*, 2006.

[37] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):231–240, Apr. 1990.

[38] G. Yalcin, O. Unsal, A. Cristal, I. Hur, and M. Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architecture PESPMA*, 2010.

[39] G. Yalcin, O. Unsal, A. Cristal, and M. Valero. FaulTM-multi: Fault Tolerance for Multithreaded Applications Running on Transactional Memory Hardware. In *Workshop on Wild and Sane Ideas in Speculation and Transactions*, 2010.

[40] G. Yalcin, O. Unsal, A. Cristal, I. Hur, and M. Valero. Symptomtm: Symptom-based error detection and recovery using hardware transactional memory. In *Parallel Architectures and Compilation Techniques (PACT), International Conference on*, pages 199–200. IEEE, 2011.

[41] D. Yoon and M. Erez. Memory mapped ecc: low-cost error protection for last level caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 116–127. ACM, 2009.

[42] Y. Zhang and K. Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Defect and Fault Tolerance in VLSI Systems, Proceedings. 18th IEEE International Symposium on*, pages 320–327. IEEE, 2003.