Combining Error Detection and Transactional Memory for Energy-efficient Computing below Safe Operation Margins

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

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

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

***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 any more. 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. According to our evaluation, using reliability schemes combined with transactional memory for error recovery reduces energy by 54 % while providing a reliability level of 100 %.**

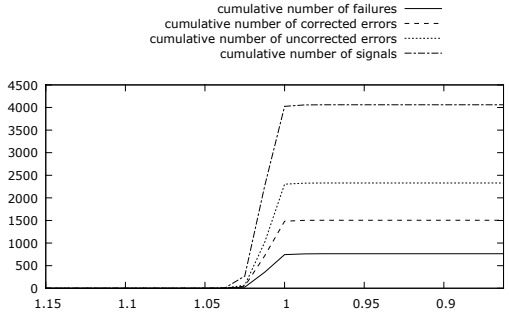


Fig. 1: Cumulative errors with maximum tolerance level

I. 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 [1]. As the power envelope becomes one of the key design concerns, 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 [2]. 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 [3]. To understand the effect of reducing V_{dd} on processors, we conduct a preliminary voltage scaling experiment in a real system having 6-core CPU AMD FX-6100. In this experiment, we start from 1.15V (i.e. nominal voltage) and reduce the CPU's V_{dd} by 12.5mV every 10 seconds (minimal voltage step for this CPU) until the system crashes. We also keep the CPU load level at approximately 99 % by performing CPU-heavy executions (i.e., MD5 calculation) on all cores. We observe faults by tracing kernel panics raised by the Machine Check Architecture (MCA) as soon as a detectable error is encountered (i.e., we set the time-out for raising kernel panic to zero) and additionally, we compare the output of the executions with the expected output. As soon as we encounter a failure, we restart the experiment from the nominal voltage. In Figure 1, we show the cumulative errors in our experiments for 838 iterations. As we can see, the lowest stable voltage for all cores of the CPU is 1.05 V. Starting from the voltage value of 1.025 V (only 10% reduction of the V_{dd}), the system starts encountering crashes within 10 seconds. More importantly, at this level the system also encounters uncorrectable errors[1]. Furthermore, during the experiment, most of the errors are encountered in the instruction cache (37%) and in the execution unit (61%), while other parts of the CPU (i.e. data cache, northbridge and combined unit) do not contribute significantly to errors distribution (less than 2%). Hence, the instruction

[1]This behavior is different from the error manifestation as described in [4], because we raise the kernel panic as soon as data corruption is detected even before it manifests to the software. Thus, a processor with only Machine Check Architecture (MCA) is not sufficient for operation below safe operation margins.

execution is highly vulnerable and requires protection to take benefit of lowering supply voltage.

Dan et al. [5] proposed Razor, a hardware solution for tolerating failures caused by lowering supply voltage. Razor extends the pipeline stages with a new circuit design in order to detect timing errors. Obviously, this circuit extension presents a hardware overhead which is not used under nominal voltage. EnerJ [6], a software framework for approximate computing, is proposed by Sampson et al. EnerJ executes the non-critical sections of applications in the low-power mode while the rest is executed under nominal voltage. Programmers specify, which parts should be approximate, thus programming language support is required. The architecture has to support separate areas in memory and CPU for which the voltage can be reduced. Our goal is seeking architectural solutions for energy minimization without requiring any substantial hardware changes or programming mechanisms.

In this 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 [7]. 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. Moreover, reliability schemes require supplementary hardware dedicated only for reliability (e.g buffers to save checkpoints) which increase system implementation and test complexity. 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 [8], but is also used for implementing reliability [9]. 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 [10] and also for the more sophisticated STAMP benchmarks [11]. Moreover, the implementation complexity of reliability schemes can be minimized by using existing TM hardware with minor changes [12]. 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 investigate the edge cases on voltage reduction while the error recovery still leads to a reduced energy consumption. In the following, we survey different existing error detection mechanisms (Section II) and introduce possibilities to combine them with TM (Section III). In Section IV we evaluate the energy reduction provided by these mechanisms.

II. Error Detection Schemes

Failures caused by decreasing the V_{dd} are either transient (i.e., soft errors and erratic bits) or persistent (yield loss and hard errors). Persistent faults can be detected by online detection schemes applied at boot time, or at the change of the supply voltage [13]. Transient failures are dynamic and therefore 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 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, 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.

A. Replication

To satisfy the strict reliability requirements of mission-critical systems, various redundancy-based error detection solutions have been proposed (e.g., [14]). 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 [12], [15], because instead of comparing each individual store instruction, one can efficiently compare the write-sets. The write-set has typically 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% (or 200% for triplication) energy and space overhead in the error-free execution. However, the energy consumption is reduced exponentially when V_{dd} is lowered, in comparison to the linearly increasing energy consumption of replication. Hence, it is worth to further investigate replication as error detection mechanism.

B. Assertions/Invariants

Using assertions is a common technique for detecting software or hardware errors [16]. 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 [17].

An approach based on a coprocessor (watchdog) is proposed in [18]. 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 [9], 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 [19] and might be inefficient because too many evaluations are needed. The alternative is to add them automatically to a program, as proposed in [20].

C. Symptom-Based Error Detection

In order to provide reliability at a low cost, some recent error detection solutions [21], [22] 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 (e.g. 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 [23] and disclosed in a patent filed by IBM [24]. In this approach, applications are executed in back-to-back, reliability purposed transactions which are 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 silent data corruptions (SDC) 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).

D. 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 [25]. 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 [26]. 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 compiler [27]. All operations must preserve the encoding, which results in more computations and higher energy consumption.

Method	Memory	Processing	Error Detection	Complexity
Replication	high	high	**high**	**low**
Assertions	medium	high	medium	high
Encoded Processing	medium	high	**high**	high
Symptoms	**low**	**low**	low	**low**

TABLE I: Comparison of error detection schemes (bold is better)

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 ANBDmem codes [28] 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 [30]. 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.

E. 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 I 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 [31]. 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. Similarly, symptom-based error detection requires only the detection of exception which already exists in the hardware. 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.

III. Error Recovery

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. 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 [32] 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 [33]. 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 [34].

Uncoordinated-local Checkpointing [35]. 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. [12]. We discuss next how TM can be used to implement error recovery.

A. 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., [12]). 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) works in two stages, a pre-commit phase and a commit phase. In the pre-commit phase the modifications are made on private copies and at the commit phase these modifications are written to the memory. 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.

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. 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. Furthermore, a rollback for redo-logging is cheap in comparison to undo-logging. Thus, if we expect high error rates, it is better to use redo-logging. Therefore, we limit the scope of this study to a previously evaluated error recovery scheme using redo-logging and do not further discuss the usage of undo-logging.

B. 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

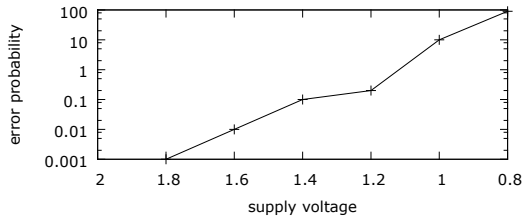


Fig. 2: Error rate as analyzed by Ernst et al. in [5].

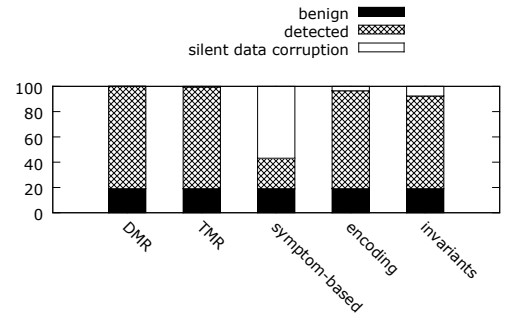


Fig. 3: Fault coverage of each scheme according to the results presented in [23], [12], [36], [29].

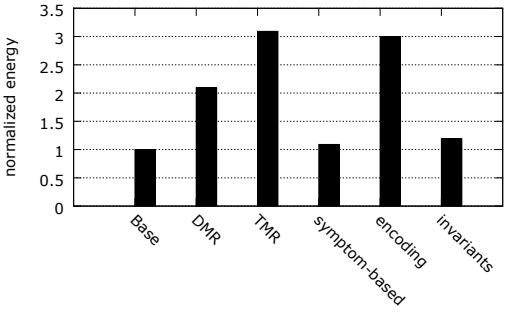


Fig. 4: Normalized energy spent by each error detection scheme when the V_{dd} is 2 V according to the results presented in [23], [12], [36], [29].

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 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 the 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).

We focus our study on light-weight transactions that are supported by hardware and target reliability rather than regular transactions with the purpose of concurrency control. Thus, they do not require code transformation and they can be committed when it is required. For instance, in regular HTMs, commit points of transactions are statically defined in the source code. On the contrary, reliability purposed transactions can be committed flexibly, for example they can commit when the HTM structures are full [12].

IV. Analysis

In this section we analyze the feasibility of applying the aforementioned error detection schemes with TM-based error recovery. We are specifically interested in how much we can lower the voltage while still providing high error detection capability. For the evaluation we consider the following scenarios: 1) We show the fault rate of execution units under scaling voltage; 2) We compare the error detection capability of each of the schemes; 3) We investigate the energy overhead of the error detection schemes and the combined error detection and recovery; 4) Finally, we consider combination of different error detection schemes.

A. Fault rate

To the best of our knowledge, only Ernst et al. performed a study in [5], evaluating the error rate of the execution units under scaling voltage. The results are shown in Figure 2. For this experiment a circuit-level design of a 64-bit Kogge-Stone adder has been implemented, assuming an 870 MHz clock and an ambient temperature of 27 C. As an input, random sample vectors are generated. The error rate is computed as the fraction of sample vectors that do not complete within the clock period of the current voltage and frequency.

B. Error detection capability

For showing the error detection capability of each scheme, we summarize results found in the literature([23] for symptom-based error detection, [12] for dual-modular redundancy (DMR), [36] for invariants and [29] for encoding) and display them in Figure 3. On average, 20 % of the injected faults are benign since they are masked before the transaction ends. Note that this is lower than the results reported in [36], because we conservatively classify faults as not being benign, if the injected fault still exists at the end of the transaction.

DMR provides 100 % fault coverage since it detects all injected faults. The coverage of TMR is slightly lower than DMR, because if there are two faulty copies, TMR can only

detect a failure, but not correct it. Note that hardware based replication (DMR or TMR) provides higher reliability since it does not have a vulnerability window as software-based replication. Symptom-based error detection only covers 35 % of the faults, if the transactions are short (i.e., less than 10,000 instructions). Similarly, encoded processing provides 97 % fault coverage, while invariants cover 93 % of the faults. Note that we include the benign faults into fault coverage and leave only the harmful data corruptions out of the coverage.

C. Energy overhead

In the following we show the energy overhead of the error detection schemes under full voltage and under scaling voltage.

Energy overhead of the error detection schemes.

We calculate the energy overhead under full voltage as the number of additional instructions (memory instructions, integer/floating point instructions) by relying on the results found in the literature ([23], [12], [36], [29]). The results are shown in Figure 4, where we normalized the values to the base, error-free energy consumption. Basically, high reliability requires more energy. For example, DMR and TMR have an overhead of more than factor two and three, respectively. The symptom-based error detection only has a negligible overhead, but also only provides a low error detection capability.

Energy consumption under scaling voltage.

For the voltage scaling experiments we use the Gem5 full-system simulator [37] and run the SPLASH2 [38] benchmark suite. The simulator runs with in-order cores executing X86 ISA running at 1 GHz with private 64KB L1D and L1I caches and a unified 2MB L2 cache. During the simulation, we inject faults to the output of each instruction with the theoretical error probability for the applied V_{dd} (given in Figure 2). Then, we assume that the faults are detected with the error detection probability as shown Figure 3 (or the fault is benign). If an error is detected, the corresponding transaction aborts and rolls back. If all transactions in the application execute without any error (but with the possibility of re-executing the transaction several times), the application completes reliably. We repeat this fault injection experiment 40 times for each application for each V_{dd} level and we calculate the reliability of the application as the rate of the executions in which the application completes reliably. To estimate the energy spent under the given V_{dd}, we calculate the total number of instructions executed (for error detection given in Figure 4 and for re-execution in the error recovery) and multiply it with the energy spent for one instruction under the given V_{dd} level.

In Figure 5 (on the left), we summarize the performance of all applications in the SPLASH benchmark by averaging their energy consumption. The energy consumption is normalized to the error-free base case in which 2 V supply voltage is used. On the right side of Figure 5 we display the averaged applications' reliability of the combined error detection and recovery under the given V_{dd}.

From these graphs we can make several observations: DMR provides the highest reliability, because it is very unlikely to have two failures affecting the replicated copies of the same instruction at the same time such that both replicated executions result in the same faulty value. However, DMR presents a high energy overhead for high V_{dd}s. When a transaction consists of 10 or 100 instructions, DMR starts to outperform the base-case, when V_{dd} is 1.4V (up to 28% reduction) or 1.2V (up to 54% reduction). For larger transactions (1,000 instructions), the overhead of DMR cannot be covered by lowering the V_{dd} anymore. Due to the increase of the transaction size the probability of faults causing rollbacks repeatedly is considerably higher. Error detection schemes other than DMR only provide a lower reliability. One reason is the high number of transactions in the SPLASH applications, which can be up to 2 billion (barnes with transaction size of 10 instructions). Hence, it is very likely that at least one transaction fails in the application.

Symptom-based error detection and invariants have a low overhead and start to outperform the base case very fast. They require up to 88% less energy than the base case for a transaction size of 10 and V_{dd}=1V. However, these schemes are not enough to complete the whole application reliably at this voltage level. When the failure rate gets very high, due to very low V_{dd} values, the error detection capabilities increase again. At these low levels it is very likely that more than one instruction per transaction fails. In this case it is easier to classify a faulty transaction, because it is only necessary to find one failed instruction to roll back. However, symptom-based error detection starts to be energy-inefficient in the moment the reliability increases.

TMR and encoded processing (schemes presenting high overhead in the base case less than 100% error detection capability), can only lead to a lower energy consumption than the base case when V_{dd} is lower than 1.4V. Since TMR does not present any recovery overhead, the supply voltage can be reduced to 0.8V (up to 80% less energy consumption than the base case). However, at these voltage levels there will be at least one transaction that produces a non-benign, faulty result.

D. Combining Error Detection Schemes

There is a tradeoff between energy efficiency and reliability, as we have seen for DMR and symptom-based error detection and TM recovery. However, there are many applications that are implicitly fault tolerant (e.g., from the area of multimedia and artificial intelligence). To reduce the overhead

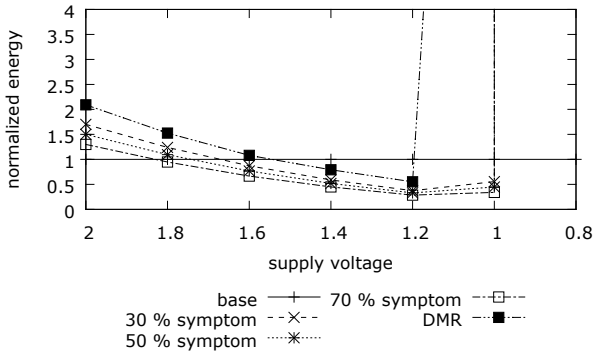
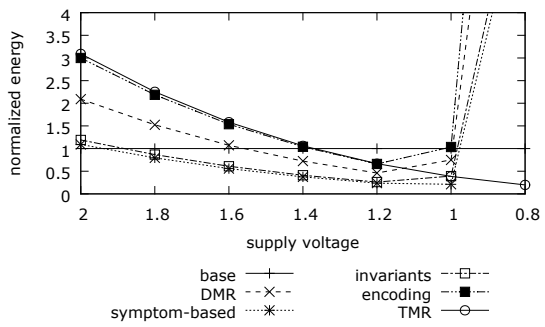
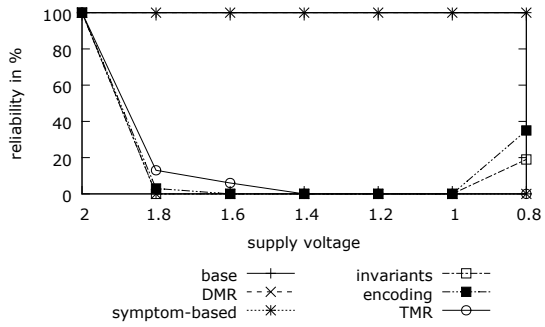


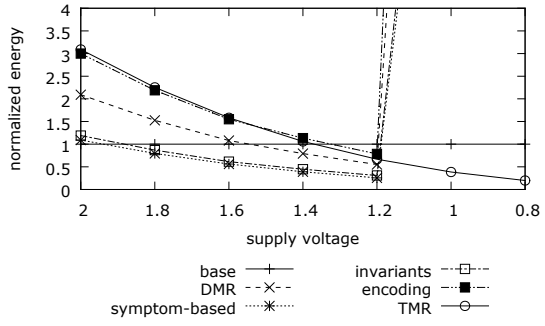
Fig. 6: Possible normalized energy consumption of the combination of DMR and symptom-based error detection (TX-Size=100 instructions).



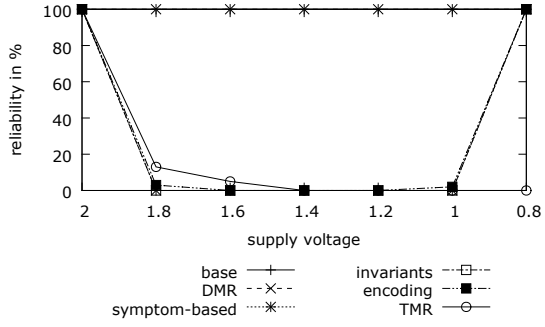
(a) Energy for transactions with 10 instructions



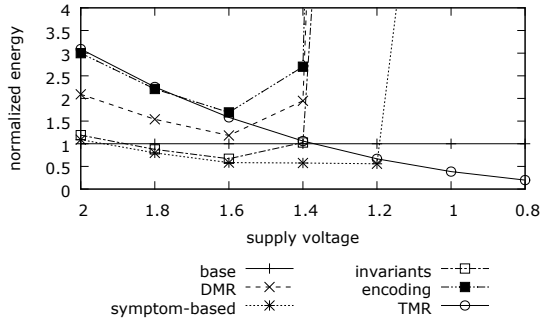
(b) Reliability for transactions with 10 instructions



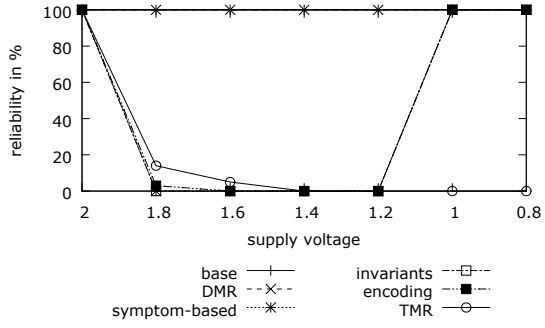
(c) Energy for transactions with 100 instructions



(d) Reliability for transactions with 100 instructions



(e) Energy for transactions with 1000 instructions



(f) Reliability for transactions with 1000 instructions

Fig. 5: Normalized energy consumption (left) and reliability (right) of SPLASH applications for different transaction sizes.

of the error detection schemes, the programmer can define parts that are less strict regarding outcome precision. Thus, we can for example combine symptom-based error detection and DMR for consuming less energy, but providing full reliability for critical parts. In Figure 6, we depict the energy overhead in comparison to the base case and DMR only for a transaction size of 100 instructions. We assume that 30, 50 or 70% of the application are only secured by symptom-based error detection. With this combination it is possible to lower the V_{dd} to 1 V (in comparison to 1.2 V with DMR only) and still be more efficient than the base case. Specifically, we reduce the energy consumption by 66% in comparison to the base case. However, at these voltage levels the reliability of symptom-based error detection is at 0%, thus might be omitted at lower voltage levels. At a voltage level of 1.6 V the reliability is around 10%, but the energy consumption 20-50% lower than the single usage of DMR.

V. 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 using 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 lowering V_{dd}.

ACKNOWLEDGMENT

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] 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*. IEEE, 2011, pp. 365–376.

[2] J. P. Kulkarni et al., "A 160 mV Robust Schmitt Trigger Based Subthreshold SRAM," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 10, pp. 2303–2313, October 2007.

[3] R. G. Dreslinski et al., "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[4] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ecc in itanium ii processors," in *International Symposium on Computer Architecture (ISCA)*, 2013, pp. 1–11.

[5] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 7–18.

[6] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[7] 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*. IEEE, 2003, pp. 320–327.

[8] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*, 2nd ed. Morgan and Claypool Publishers, 2010.

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

[10] 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*, ser. ISLPED '05, 2005, pp. 331–334.

[11] 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*. USENIX Association, 2012, pp. 6–6.

[12] G. Yalcin, O. Unsal, and A. Cristal, "FaulTM: Fault-Tolerance Using Hardware Transactional Memory," in *Design, Automation and Test in Europe DATE*, 2012.

[13] 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*, 2007, pp. 97–108.

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

[15] G. Yalcin, O. Unsal, and A. Cristal, "Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory," in *International Conference on Computing Frontiers*, 2013.

[16] D. Andrews, "Using executable assertions for testing and fault tolerance," in *9th Fault-Tolerance Computing Symp*, 1979, pp. 20–22.

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

[18] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors-a survey," *Computers, IEEE Transactions on*, vol. 37, no. 2, pp. 160–174, 1988.

[19] 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*, vol. 16, no. 4, pp. 432–443, 1990.

[20] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.

[21] 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.

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

[23] 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*. IEEE, 2011, pp. 199–200.

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

[25] D. Yoon and M. Erez, "Memory mapped ecc: low-cost error protection for last level caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 116–127.

[26] P. Forin, "Vital coded microprocessor principles and application for various transit systems." in *IFAC/IFIP/IFORS Symposium*, 1989, pp. 79–84.

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

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

[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: IEEE Computer Society, 2010.

[30] J.-T. Wamhoff, M. Schwalbe, R. Faqeh, C. Fetzer, and P. Felber, "Transactional encoding for tolerating transient hardware errors," in *Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium (SSS 2015)*, ser. SSS '13, T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, and M. Yamashita, Eds. Springer International Publishing, November 2013, vol. 8255, pp. 1–16.

[31] 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*. IEEE, 2008, pp. 70–79.

[32] 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, FTCS-25*, 1995, pp. 289 –298.

[33] 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*, jun 1990, pp. 82 –88.

[34] 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*, ser. ISCA '11, 2011, pp. 153–164.

[35] 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.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[36] J. Chang, G. Reis, and D. August, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks*, 2006, pp. 83–92.

[37] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *SIGARCH Compututer Architecture News*, vol. 23, pp. 24–36, May 1995.