Computers and Electrical Engineering 36 (2010) 480–490

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Improving chip multiprocessor reliability through code replication ☆

Ozcan Ozturk

Computer Engineering Department, Bilkent University, 06800 Bilkent, Ankara, Turkey

ARTICLE INFO

Article history:
Received 11 January 2009
Received in revised form 5 September 2009
Accepted 23 November 2009
Available online 4 January 2010

Keywords:
Reliability
Code replication
Energy consumption
Chip multiprocessors
Compilers

ABSTRACT

Chip multiprocessors (CMPs) are promising candidates for the next generation computing platforms to utilize large numbers of gates and reduce the effects of high interconnect delays. One of the key challenges in CMP design is to balance out the often-conflicting demands. Specifically, for today's image/video applications and systems, power consumption, memory space occupancy, area cost, and reliability are as important as performance. Therefore, a compilation framework for CMPs should consider multiple factors during the optimization process. Motivated by this observation, this paper addresses the energy-aware reliability support for the CMP architectures, targeting in particular at array-intensive image/video applications. There are two main goals behind our compiler approach. First, we want to minimize the energy wasted in executing replicas when there is no error during execution (which should be the most frequent case in practice). Second, we want to minimize the time to recover (through the replicas) from an error when it occurs. This approach has been implemented and tested using four parallel array-based applications from the image/video processing domain. Our experimental evaluation indicates that the proposed approach saves significant energy over the case when all the replicas are run under the highest voltage/frequency level, without sacrificing any reliability over the latter.

1. Introduction

Chip multiprocessors (CMPs) are promising candidates for the next generation computing platforms [16]. They have already started to enter the marketplace over the past five years or so and have taken considerable attention from both academic and industry circles. While many of the compilation techniques developed in the context of high-performance parallel computing can be adapted to work with the CMP architectures as well, achieving high-performance through parallelism is not the only issue in this computing domain. Specifically, for today's image/video applications and systems, power consumption, memory space occupancy, area cost, and reliability are as important as performance. Therefore, a compilation framework for CMPs should consider multiple factors during the optimization process.

Reliability issues in the existence of transient errors are becoming an increasingly critical challenge. For example, recent research [5,10,15] has underlined the importance of protection mechanisms against soft errors (a form of transient errors) that are caused by particle strikes. While one can potentially implement costly mechanisms in hardware to cope with the problem of transient errors, such an approach would not be suitable, where, as mentioned above, multiple factors should be balanced out carefully. In particular, while optimizing for reliability, one needs to consider issues such as performance, power and memory overheads.

☆ This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme. A two page embedded system reliability improvement technique was presented in SOCC 2006 conference proceedings. This submission differs from our previous work by extending it to general purpose chip multiprocessors, by presenting architectural description and algorithmic details of the approach, by giving an example of code replication, and by presenting an extensive experimental evaluation of the idea.
E-mail address: ozturk@cs.bilkent.edu.tr

0045-7906/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2009.11.004

In the past, one of the ways of improving transient error detection capabilities has been data/code replication. While from the performance angle it might be possible to hide the additional latencies introduced by replication (e.g., in a CMP, one can execute extra computations in parallel with main computation if there are available resources), power implications are more difficult to mitigate and very concerning for battery-operated embedded systems. This paper proposes a *compiler-directed* energy-aware code replication scheme for CMP architectures. There are two main goals behind our compiler approach:

- We want to minimize the energy wasted in executing replicas when there is no error during execution (which should be the most frequent case in practice).
- We want to minimize the time-to-recovery (through the replicas) from an error when it occurs.

Our compiler-directed scheme achieves this by executing replicas with scaled down voltage/frequency.[1] The replicas and the primary copies start executing at the same time. In this case, when no error occurs (which is determined by the successful termination of the primary copy), we terminate the replica and since it has operated with lower voltage/frequency so far, we save energy, compared to the case where the replica is executing with the highest voltage/frequency available. On the other hand, when an error occurs in the primary copy, it is aborted, and the replica is switched to the highest voltage/frequency level to minimize the time to complete the task. This approach has been implemented within an optimizing compiler and tested using four array-based applications from the image/video processing domain. Our experimental evaluation indicates that this approach saves significant energy over the case when all the replicas are run under the highest voltage/frequency level, without sacrificing any reliability over the latter. Based on our experiments, we recommend the compiler writers for the CMP architectures to incorporate energy-aware code (computation) replication into their suite of optimizations.

The rest of this paper is organized as follows. Section 2 gives a brief description of the CMP architecture considered in this paper. Section 3 describes the problem attacked in more detail and Section 4 gives our compiler based solution. The experimental methodology and the benchmark codes used in the experiments are given in Section 5. Section 6 discusses the related work and the paper is concluded with a brief summary in Section 7.

2. CMP architecture

The CMP architecture we focus in this paper is a shared memory based one, wherein each processor has its private L1 instruction and data caches and a shared on-chip unified L2 cache. While our approach can potentially work with any interconnection topology, in this work, we assume a simple bus based on-chip network and a MESI-like coherency protocol is employed. We also assume a large off-chip main memory, which can be used to store instructions and data. Clearly, exploiting the L1–L2 cache hierarchy is critical in this CMP architecture and we assume that the codes we target have already been optimized for data locality.

We assume that, in this CMP architecture, each processor can be operated under different voltage/frequency levels. That is, the voltage/frequency of each processor can be set to a different level (from among a set of hardware-supported levels) than the others. While dynamically switching from one voltage/frequency level to another during execution takes time and energy, in our approach, these transitions do not occur very frequently. In any case, the experimental data presented in this paper include these energy/performance overheads as well. Note that, memories are exempt from these voltage/frequency switches as they require additional constraints.

3. Problem description and overview of our approach

Before discussing our approach, let us point out the main problems associated with two more straightforward approaches. Suppose that, a given loop nest in an application is parallelized over k out of a total p processors in our CMP. We use the term *primary copy* for the code assigned to each of the k processors. If $p - k \geqslant k$, we can create k *replicas* (one for each primary). On the other hand, if $p - k \leqslant k$, we create only $p - k$ replicas; the remaining primary copies $(2k - p)$ are run without replicas. In either case, in a straightforward approach, the replicas are executed in parallel with the primaries using the same (highest) voltage/frequency level as the primaries. If a primary fails, we use the result of the replica if it terminates successfully. In practice, the cases where both a primary copy and its replica fail should be really rare. The main problem of this straightforward scheme from the power perspective is that replicas can consume a significant amount of power since they use the highest voltage/frequency level. Consequently, the energy overheads incurred by this replication scheme can be very large in practice and, most of the time, this energy is simply wasted. However, the recovery time when the primary copy fails is fast (provided that the replica executes correctly) since the replica starts its execution at the same time with the primary copy.

An alternate and easy-to-implement approach would be starting to execute a replica only if (after) the corresponding primary copy fails. As compared to the scheme described in the previous paragraph, this approach is preferable from the energy angle. This is because it does not consume any extra energy (in executing replicas) as long as no errors occur. However, the

[1] Scaling supply voltage leads to quadratic power savings. Due to issues related to signal integrity and reliability, supply voltage and clock frequency should be scaled together.

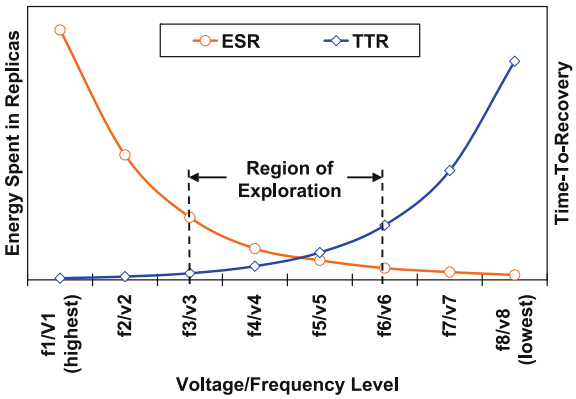

Fig. 1. Variation of ESR and TTR with respect to voltage/frequency level.

recovery time in this approach can be very high since the replica starts execution very late (i.e., only after the primary copy fails). To summarize, while the first scheme explained above is preferable from the time-to-recovery perspective, the second one is preferable from the power consumption perspective.

The goal behind the scheme proposed in this paper is to strike a balance between the two extreme schemes described above, and therefore, provide a designer with more options she can choose from. More specifically, we want to reduce energy consumption in replicas as much as possible in the case where no errors occur. At the same time, we want to reduce the time required to recover from an error as much as possible when it occurs. We achieve these objectives by executing the replicas with reduced voltage/frequency when the resulting latency can be tolerated. The idea is plotted in Fig. 1 for an example case. Each point on the *x*-axis in this figure corresponds to a different supply voltage/clock frequency pair under which a replica is run (the primary copies always use the highest voltage/frequency level available). The *energy-spent-in-replicas* (*ESR*) curve represents a typical trend for the amount of energy spent in a replica up until the point the associated primary copy has finished its execution successfully or it has signaled an error and aborted. As we reduce the voltage and frequency values, we witness a reduction in the ESR value as well. The *time-to-recovery* (*TTR*) curve, on the other hand, represents the time it takes to recover, i.e., the time it takes for the replica to finish its execution after the primary copy signaled the error. As we can see, the TTR curve exhibits an opposite trend compared to the ESR curve. As we reduce the value of voltage/frequency pair, the TTR value gets increased. In general, one would want to impose some bounds for the acceptable ESR and TTR values and these two bounds define a *region of exploration* as far as the voltage/frequency values to be used for replicas are concerned. In the example plot in Fig. 1, it is assumed that this region of exploration is between f3/v3 and f6/v6. The goal behind our experimental evaluation is to study the impact of a selected voltage/frequency value on the values of ESR and TTR for a set of applications.

Fig. 2 illustrates, at a high level, how our approach operates when the execution of the primary copy is error-free (a) and when there is a transient error (b). In the error-free case shown in Fig. 2a, the primary copy sends a signal (after completing its execution) to the corresponding replica to stop executing (to save energy), whereas if an error is detected in the execution of the primary copy, the primary copy terminates its execution and sends a signal to replica to speedup in order to minimize the performance penalty by finishing the execution of the replica as soon as possible (see Fig. 2b). Note that, switching from one voltage/frequency level to another one may increase the possibility of causing an error on the replica, however, in this paper, we do not deal with the errors in replicas. As stated earlier, in practice, the cases where both a primary copy and its replica fail should be really rare. We also want to emphasize that our approach replicates computations only when there are available processors to execute replicas. Therefore, the replicas are executed in parallel with the primary copies, and consequently, their impact on overall performance is minimum (less than 2% on average in our experimental evaluation).

Algorithm 1. Matrix multiplication

```
1: Input: Matrices A and B
2: Output: Matrix C
3: for i ← 1...N do
4:   for j ← 1...N do
5:     for k ← 1...N do
6:       C[i][j]+ = A[i][k] * B[k][j]
7:     end for
8:   end for
9: end for
```

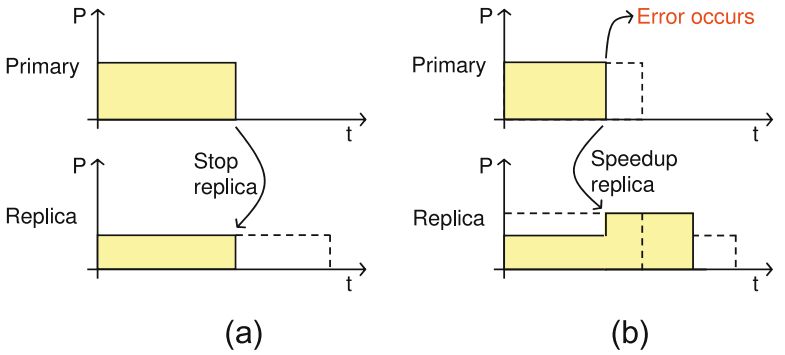

Fig. 2. An example scenario. (a) Error-free execution. (b) Execution with a transient error. For each plot, the *x*-axis represents power (voltage level). Note that the replica is executed under lower voltage/frequency level until the primary finishes successfully, or it fails.

Table 1
Possible invariants for matrix multiplication.

	Invariant
1	Size(A) == size(B)
2	Size(A) == size(C)
3	Elements of A, B, and C != null
4	A, B, and C does not contain any duplicates
5	Address(B[i_1][i_2]) = address(A[i_1][i_2]) + D_1
6	Address(C[i_1][i_2]) = address(A[i_1][i_2]) + D_2
7	Loop indices i, j, $k \geqslant 0$

4. Details of our approach

The compiler has two roles in our approach. First, it is used to parallelize the given sequential application to be executed on a CMP. Our current implementation employs a parallelization strategy which tries to achieve outermost loop parallelism for as many loop nests in the application as possible. The second role of the compiler is to create replica computations and insert coordinating code in both replica and primary copy to enable switching between them at runtime.

There are several details of the proposed approach that need to be discussed. One of these is regarding the interaction between the primary copy and the replica. In particular, the primary copy should be able to detect whether it fails or succeeds and interact with the corresponding replica (if one exists) accordingly. One way of implementing this is to use loop invariants.[2] Let us consider the matrix multiplication code given in Algorithm 1 as an example. This program takes two two-dimensional arrays A and B as input, and returns their product in the output array C. It contains a set of nested loops indexed by variables i, j and k. Table 1 shows some of the possible loop invariants which could be detected by a dynamic invariant detector such as Daikon [11]. This tool detects the likely invariants over the programs data structures. The invariants shown are generated based on the C version of the code, where arrays A, B, and C have N^2 elements each. Based on the invariants given in Table 1, one can see that some of these invariants are easy to check, whereas some others are more difficult (costly). For example, the invariants that compare loop indices can easily be checked, whereas checking whether array B contains no duplicates is very costly at runtime.

The original matrix multiplication code given in Algorithm 1 is modified by the compiler automatically using the invariants given in Table 1. The purpose of these modifications is to signal the existence of a transient error when one of the invariants fail. The modified code, given in Algorithm 2, first checks whether the invariant conditions hold. In this example, for clarity of presentation, we only check the values of the loop indices and the addresses of the matrices based on the last three invariants (i.e., invariants 5, 6, and 7). This modified code represents the primary copy where possible errors are detected. If there are no errors during the execution, then the replica is terminated to save energy (line 29). However, if a transient error is detected (based on the invariants),[3] the replica is informed by the *speedup* signal right before the primary stops itself running with the *exit( )* command. In the primary copy, the addresses of arrays A, B, and C are checked before entering the k loop. Similarly, the index values i and j are also validated right before the k loop. Although the replica runs under a low voltage/frequency

[2] A loop invariant is an assertion that holds true each time a loop condition is tested [1].

[3] It is important to note that using loop invariants provide only a certain level of error detection capability. It is perfectly possible that a loop invariant signals no error while the application is actually suffering from a transient error. Our goal here is just to illustrate how the primary copy and the replica interacts. Our approach can easily be made to work with other error detection approaches [13,17,20] as well.

level when there are no errors, it continuously checks for signals (could be speedup or terminate) from the primary copy. Once the replica receives a signal (in the innermost loop), it acts accordingly; i.e., it either terminates or switches to the highest voltage/frequency level available. Algorithm 3 shows the code of the replica for the matrix multiplication case. Note that it checks for signals coming from the primary copy within the innermost loop to be able to respond the situation quickly.

Algorithm 2. Reliable matrix multiplication (primary)

```
1:  Input: Matrices A and B
2:  Output: Matrix C
3:  for i ← 1...N do
4:    for j ← 1...N do
5:      if addr(B[i][j]) ≠ addr(A[i][j]) + D1 then
6:        speedup(replica, Address error : A, B)
7:        exit()
8:      else if addr(C[i][j]) ≠ addr(A[i][j]) + D2 then
9:        speedup(replica, Address error : A, C)
10:       exit()
11:     else if i < 0 or i > N then
12:       speedup(replica, Index error : i)
13:       exit()
14:     else if j < 0 or j > N then
15:       speedup(replica, Index error : j)
16:       exit()
17:     else
18:       for k ← 1...N do
19:         if k < 0 or k > N then
20:           speedup(replica, Index error : k)
21:           exit()
22:         else
23:           C[i][j]+ = A[i][k] * B[k][j]
24:         end if
25:       end for
26:     end if
27:   end for
28: end for
29: terminate(replica)
```

Algorithm 3. Reliable matrix multiplication (replica)

```
1:  Input: Matrices A and B
2:  Output: Matrix C
3:  for i ← 1...N do
4:    for j ← 1...N do
5:      for k ← 1...N do
6:        if receive(speedup) then
7:          Processor_Frequency ← f_max
8:          Processor_Voltage ← v_max
9:        else if receive(terminate) then
10:         exit()
11:       end if
12:       C[i][j]+ = A[i][k] * B[k][j]
13:     end for
14:   end for
15: end for
```

An important issue to discuss at this point is the question of what happens when there is some data dependency between the different iterations of a loop nest. If two instances of such a data dependent code are running on the same data, this might cause incorrect execution based on the type of the dependency taking place and the set of array elements involved. To illus-

$$\begin{array}{l}\text{for } i = 2 \text{ to } N-1\\ \quad A[i] = A[i+1] * A[i-1]\end{array}$$
(a)

$$\begin{array}{l}A' \leftarrow A\\ \text{for } i = 2 \text{ to } N-1\\ \quad A[i] = A[i+1] * A[i-1]\end{array}$$
(b)

$$\begin{array}{l}\text{for } i = 2 \text{ to } N-1\\ \quad A'[i] = A'[i+1] * A'[i-1]\end{array}$$
(c)

Fig. 3. An example code with data dependencies.

trate this, let us consider the example given in Fig. 3a, there is a flow-dependence between $A[i]$ and $A[i-1]$ and an anti-dependence between $A[i]$ and $A[i+1]$. Assume a specific scenario where the primary copy runs for k iterations before the invariants signal an exception. Since the replica is running on a lower voltage/frequency level at the point where the primary copy fails, it should be executing iteration m, where $m < k$. Once the primary copy stops executing, the replica will continue to run on the same data, which will cause $A[m+1] \ldots A[k]$ to be computed incorrectly. One possible solution to this problem, which is also the one adopted in our current implementation, is to replicate data for the replicated code if there is any such dependency across loop iterations. In our current example, instead of the replica running on array A, it can compute its results based on A', which needs to be initialized with the original values of A. Fig. 3b shows the primary copy and Fig. 3c gives the replica for this last example. While this certainly introduces some memory overhead, we found that this overhead was not very much (less than 5% on the average across our applications).

5. Experimental evaluation

Our implementation uses the SUIF compiler infrastructure [2] for generating the primary and replica computations. The additional compilation overheads brought by our approach was less than 55% for all applications tested, including the time spent in Daikon for detecting loop invariants. We performed our experiments using the SIMICS [19] simulation tool. SIMICS is a simulation environment for multiprocessor environments that can perform cycle accurate timing modeling, do hardware verification, and use micro-architectural interfaces for processor and memory subsystem architecture design work. We enhanced the SIMICS environment to embed energy models (to compute energy) and simulated a CMP architecture. The energy models used for CPU and interconnect are from the Wattch tool-set [6]. In this tool-set, power estimates are based on a suite of parameterizable power models for different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation. Dynamic power consumption depends on load capacitance (C), supply voltage (V_{dd}), clock frequency (f), and activity factor (a), where a is a fraction between 0 and 1 indicating the switching activity on average. More specifically, C is estimated based on the circuit and the transistor sizings, whereas V_{dd} and f depend on the process technology. On the other hand, the activity factor – a – is related to the application being executed. The capacitances of the units of a processor are key to the overall power consumption. These individual units are reduced into stages, where delay for the entire part is estimated using stage delays. The energy models for the memory components on the other hand are based on CACTI [34]. In this model, energy is modeled as $E = C_L \times V_{dd}^2 \times P_{0\to1}$, where C_L is the physical capacitance and $P_{0\to1}$ is the probability that the device will consume energy. Technology size given as a parameter scales the capacitances and the V_{dd} value. A memory component is divided into stages, where physical capacitance of each stage is used to calculate the power consumption. To perform our experiments, we also implemented an error injection module within the SIMICS environment that can inject errors into the memory components in our CMP architecture. For every memory instruction executed, we invoke an error injection function to inject errors into the lines in main memory and caches. The error injection function scans the memory and caches; every bit has a fixed probability of incurring an error. We can think of this process as flipping a coin for every bit, which has a fixed probability of having an error. Once a bit incurs an error, this error's address is recorded by the

Table 2
Our benchmark codes.

Benchmark name	Input size (KB)	Average number of processors	ESR (nJ)	TTR (μs)
Rawaudio	376.33	5.11	377.18	543.34
Md5	186.88	4.28	54.91	210.05
Cordic	193.07	4.48	63.57	196.51
G721	822.41	5.92	574.62	308.95

Table 3
Our major simulation parameters for the CMP architecture and their default values.

Parameter	Values
Number of processors	8
Number of voltage/frequency levels	8
Highest/lowest voltage level	1.4 V/0.7 V
Frequency step size	30 MHz
Voltage/frequency transition penalty	10 cycles/2.10 nJ
Private on-chip L1 data/instruction cache	8 KB; 2-way; 32 byte line size; 2 cycle latency
Shared on-chip L2 cache	512 KB; 4-way; 128 byte line size; 12 cycle latency
On-chip interconnect arbitration latency	4 cycles
Off-chip memory	64 MB; 110 cycles latency
Error injection rate	10^{-9}

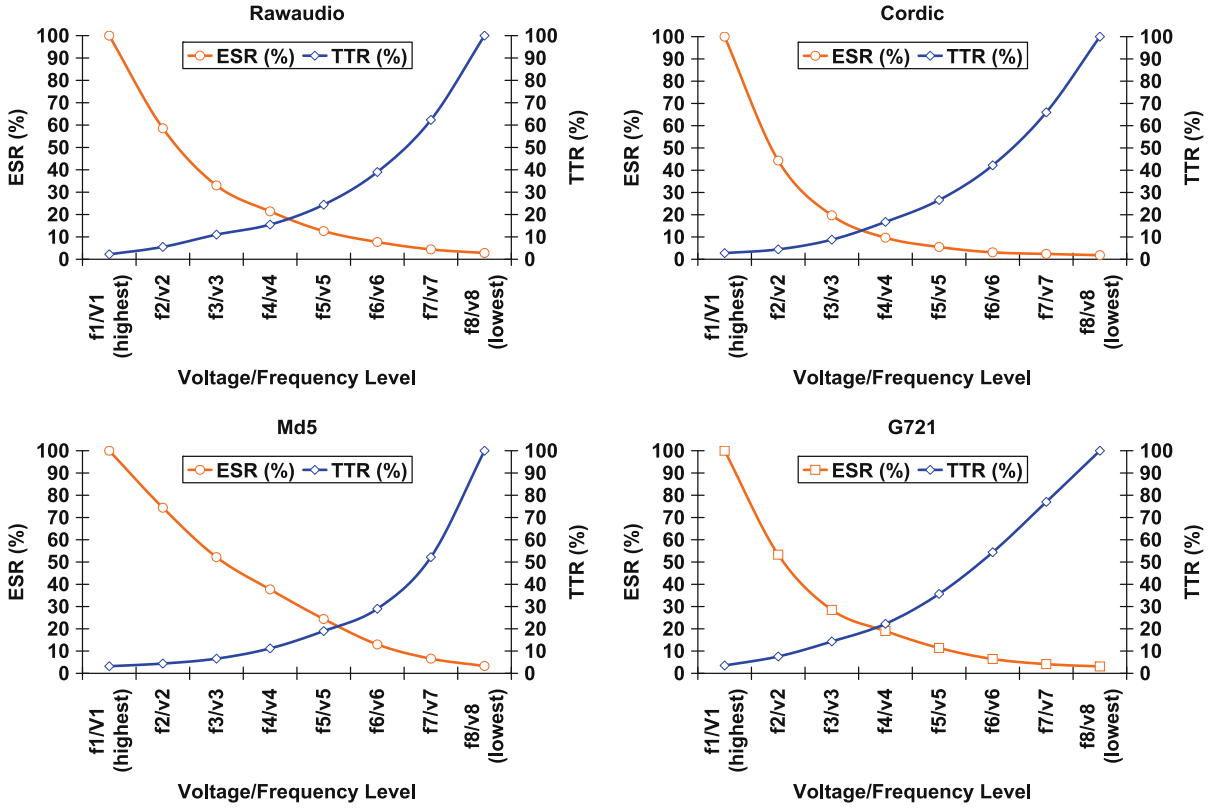

Fig. 4. ESR and TTR values (normalized) with different voltage/frequency levels for replicas.

error injection module. The default value for the error injection probability for our base experiments was 10^{-9}. It must be emphasized that the error injection rates used in our experiments are much more aggressive than those in current technologies [5,21,33] to better reflect future technologies and better capture the higher number of errors that will occur in longer-running applications. Note also that many applications such as those employed in ATMs, industrial microcontrollers, and automobiles are long running and reliability-critical.

The set of benchmark codes used in this study are given in Table 2. These codes are randomly selected from the MediaBench suite [18]. The third column gives the number of processors used to execute a loop nest, when averaged over all loop nests in the application. Note that each nest is executed using the ideal number of processors from the performance perspective. That is, increasing the number of processor any further does not bring any additional performance benefits, and the processor efficiency, as defined in [14], drops. Since we have eight processors in our default configuration (see Table 3), we see from this third column that, for all the benchmarks, we have some idle processors that can be used for replicating computation. The values in the last two columns are obtained using the default simulation parameters in Table 3. The fourth column of Table 2 gives the ESR value for each application when averaged across all the loop nests in the application. The last

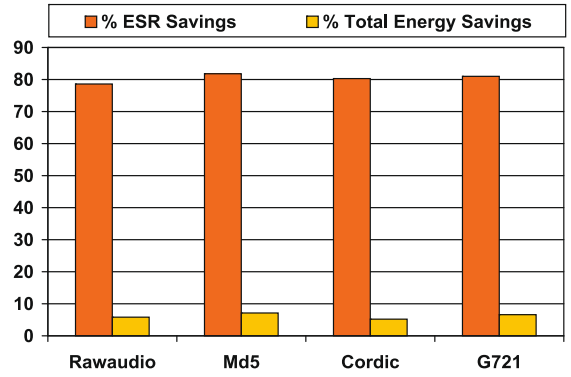

Fig. 5. Energy savings when 20% increase in TTR can be tolerated.

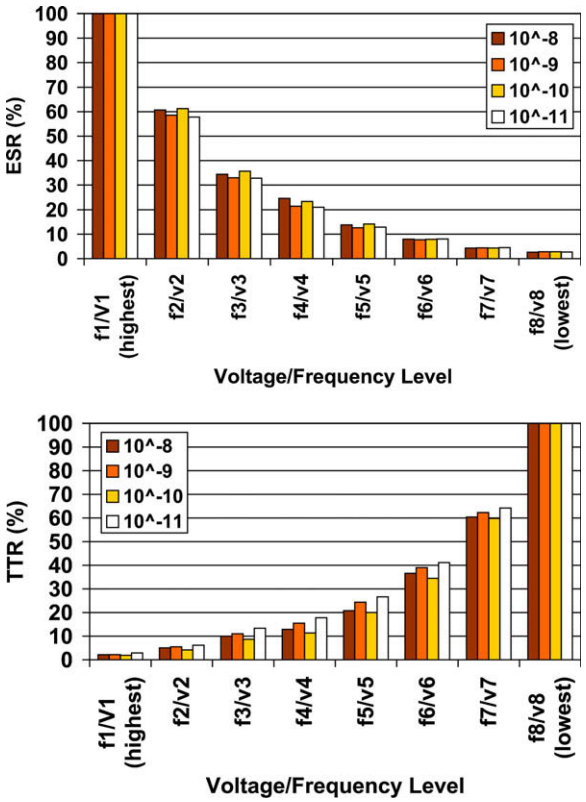

Fig. 6. Results with different error injection rates for Rawaudio.

column gives the TTR value for each application, again when averaged over all the loop nests in the application. The values in the last two columns of this table are obtained by injecting transient errors into the memory components[4] under the default injection rate of 10^{-9}. The ESR values are collected by running the replicas with the highest voltage/frequency level available. In contrast, the TTR values are obtained by running the replicas under the lowest available voltage/frequency level. The results presented in the rest of this section are given as a fraction of the values in these last two columns of Table 2. We also want to mention that the increase in the data memory requirements due to our approach was less than 5% on the average. Finally, since our approach runs replicas along with the primaries in parallel, the impact of execution time of the original applications was minimal. More specifically, we found that our energy-aware code replication scheme causes about 2% performance penalty

[4] Our approach can work when errors are injected into other components as well. The reason that we focus on error injected into the memory components is the fact that our current error injector only works for memory components.

when averaged over the codes in our experimental suite. This overhead is mainly due to data copying to cope with dependences between the primaries and replicas and due to synchronization of replicas and primaries.

Fig. 4 gives the ESR and TTR values when the replicas are executed under different voltage/frequency levels. From these plots, we can clearly see the tradeoff between ESR and TTR. Specifically, if we want to reduce the ESR value, we need to execute the replicas with low voltage/frequency. In comparison, if we want to reduce the TTR value, we might want to use a high voltage/frequency level for the replicas. This observation holds for all four benchmarks in our suite. Clearly, if both these metrics (ESR and TTR) are important to a given application (which should normally be the case), one should refrain from employing the highest or lowest voltage/frequency levels for replicas. By analyzing these plots, a designer can easily make the appropriate tradeoffs between ESR and TTR. Typically, the designer would have some bounds on ESR and TTR. The bound on ESR requires the voltage/frequency level to be below a certain value (on the *x*-axis of Fig. 4), whereas the bound on TTR requires the level to be above a certain value. Within this region, the designer can explore the tradeoffs between ESR and TTR and select the most suitable voltage/frequency level for her design.

To illustrate the potential savings that can be achieved through our approach, the bar-chart in Fig. 5 gives the energy savings when we can tolerate 20% increase in the TTR value as compared to the case where we employ the highest voltage/frequency level in executing the replicas. From the first bar of each benchmark, we see that the average ESR saving across the four benchmarks is around 80.3%. The second bar, for each benchmark, gives the total energy savings in this case (considering CPU, interconnects, and memory components). The average saving in total energy is 6.19%, which shows that our approach is beneficial when we consider the total energy consumption as well.

Fig. 6 gives the results for one of our benchmarks, Rawaudio, under the different error injection rates. Recall that the default error injection rate used in the experiments so far was 10^{-9}. The trends with the remaining three benchmarks are similar; so, they are not presented here in detail. Our main observation from these results is that the behavior of our approach does not significantly change with varying error injection rates. This is because the results presented in Fig. 6 are normalized results, as mentioned earlier. We can conclude from these results that our approach performs well across the different error injection rates. Note that, ESR and TTR values do not monotonically increase/decrease with the increasing error injection rates. This follows from the fact that, a change in the error injection rate causes a change in the remaining computation for the replica. This remaining computation either increase or decrease depending on the error injection rate, thereby increasing/decreasing the ESR and TTR values. However, when normalized with respect to the lowest/highest voltage level, the normalized ESR/TTR values may not follow the same order as the error injection rates (since proportions are used).

6. Related work

Reliability issues in the existence of transient errors are becoming an increasingly critical challenge for emerging designs. Previous research explored both hardware and software mechanisms to cope with the transient errors. Mahmood and McCluskey [20] describe Watchdog processors upon which most of the following fault detection architectures are based. The SMT machines that allow multiple independent threads to execute simultaneously are used for fault detection. Rotenberg suggest using an SMT architecture to execute two copies of the same program for resilience against faults [27]. In this approach, two copies of the same program execute with a lag and the outcomes of the two threads is verified. This approach has been enhanced in Slipstream [29]. Reinhardt and Mukherjee [26] propose Simultaneously and Redundantly Threaded (SRT) processors that perform replication in hardware to provide transparent and continuous fault coverage. In comparison, Vijaykumar et al. [32] propose a scheme called Simultaneously and Redundantly Threaded processors with Recovery (SRTR) that further enhances SRT. Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) is proposed by Gomaa et al. [12]. CRTR extends SRTR for transient-fault recovery in SMT to perform well on-chip multiprocessors. In the DIVA architecture [3], Austin augments the commit phase of the processor pipeline with a functional checker unit. This simple in-order checker processor verifies the output of the complex out-of-order processor, and permits only the correct results to commit. Ray et al. [25] propose an extension to superscalar out-of-order architectures. In this extension, the results from the replicated threads are checked for correctness. In case of an error, they employ an error-recovery scheme that rewinds the instructions to restart at a failed instruction. When a fault is detected, Tamir and Tremblay [30] use micro rollback to recover the microarchitecture state. Introspection [17], proposed by Karri and Iyer, implements a register transfer level technique for fault-tolerance. It exploits idle computation cycles in the datapath and idle data transfer cycles in the interconnection network to detect errors. Another hardware based approach is proposed by Zhang [35], in which a fully-associative cache is added to the architecture. This cache stores the replica(s) of every write to the L1 data cache to detect and correct soft errors.

Software techniques proposed so far are mainly based on replicating code and/or data. Replicating the computations is a common technique which could be carried out at different granularities, ranging from procedure level to statement level. In [28], the authors propose processor level replication, whereas the work discussed in [23] operates at a transaction level. Similarly, Cooper [9] employs a procedure level replication, and Balasubramanian and Banerjee [4] implement statement level replication. Gong et al. [13] extend the work proposed in [4], where they employ a compiler supported approach to fault detection in regular loops executing on distributed-memory systems. In this work, the select instances of loop statements are replicated to ensure the specified fault tolerance coverage. Narayanan et al. [22] propose using loop invariants to detect soft errors. In this approach, at every *i* iterations of a loop nest, they check whether the loop invariants hold. Note that our approach can use many of these techniques in the primary copy to detect errors, though in our experiments, we used only

the loop invariant based error detection. In addition, our approach corrects errors through the replicas in an energy-sensitive manner. Unsal et al. [31] discuss an energy-aware, software-based fault tolerance approach targeting at real-time systems. While the concerns of both our paper and the work in [31] are similar, we focus on the compiler-directed generation of replicas. Another approach presented in [7] tries to improve reliability against transient errors for embedded chip multiprocessors. Specifically, they either utilize the idle processors by duplicating the computation, or place them into the low-power mode to save energy. In this scheme, either energy is saved or reliability is improved for a given loop nest. On the other hand, in our approach, we reduce energy consumption in replicas as much as possible in the case where no errors occur, while improving reliability. Rashid et al. [24] present a thread-level redundancy (TLR) approach where a lead processor executes and multiple checker processors perform parallel verification. In order for the checkers to keep up with the lead processor, authors parallelize the workload by dividing the dynamic instruction stream into chunks of consecutive instructions, and distribute different chunks to multiple checkers for parallel verification. Note that, this approach requires multiple checker processors since they are running at a lower voltage level. However, in a chip multiprocessor environment, there might be limited number of processors available to perform parallel verification. Moreover, there is a huge communication overhead when lead processor passes checkpoint information to the checker processors at every checkpoint. In an extended abstract [36], Zhu and Aydin discuss the tradeoff with different redundancy granularities ranging from thread-level duplication to process-level duplication. The only other code replication and reliability related compiler work we are aware of is [8], which is an energy-aware reliability improvement technique for embedded systems.

7. Conclusion

The main contribution of this paper is a compiler-directed energy-aware computation replication scheme for the CMP (chip multiprocessor) architectures. Our primary goal is to improve reliability with as little energy and performance overheads as possible. The main idea behind our compiler-directed approach is to run the replicated computations with reduced voltage/frequency, thereby striking a balance between the time to recover (when an error occurs) and the energy wasted in replicated computations. We tested our approach with four image/video applications by simulating their executions and injecting transient errors during execution. The experimental results we obtained are very encouraging in the sense that they show our approach saves significant amount of energy with respect to a straightforward computation replication scheme that runs replicas using the highest voltage/frequency level available.

References

[1] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. Boston, MA, USA: Addison-Wesley Longman Publishing Co.; 1986.
[2] Amarasinghe SP, Anderson JM, Lam MS, Tseng CW. The SUIF compiler for scalable parallel machines. In: Proceedings of the seventh SIAM conference on parallel processing for scientific computing; February 1995.
[3] Austin TM. Diva: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd annual ACM/IEEE international symposium on microarchitecture; 1999. p. 196–207.
[4] Balasubramanian V, Banerjee P. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. IEEE Trans Comput 1990;39(4):436–46.
[5] Baumann R. Soft errors in advanced computer systems. IEEE Des Test 2005;22(3):258–66.
[6] Brooks D, Tiwari V, Martonosi M. Wattch: a framework for architectural-level power analysis and optimizations. In: Proceedings of the 27th annual international symposium on computer architecture; 2000. p. 83–94.
[7] Chen G, Kandemir MT, Li F. Energy-aware computation duplication for improving reliability in embedded chip multiprocessors. In: ASP-DAC; 2006. p. 134–9.
[8] Chen G, Ozturk O, Chen G, Kandemir M. Energy-aware code replication for improving reliability in embedded chip multiprocessors. In: Proceedings of the IEEE international SOC conference (SOCC'06), Austin, TX; September 2006.
[9] Cooper EC. Replicated distributed programs. In: Proceedings of the 10th ACM symposium on operating systems principles; 1985. p. 63–78.
[10] Degalahal V, Vijaykrishnan N, Irwin MJ. Analyzing soft errors in leakage optimized SRAM design. In: Proceedings of the 16th international conference on VLSI design; 2003.
[11] Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. IEEE Trans Softw Eng 2001;27(2):99–123.
[12] Gomaa M, Scarbrough C, Vijaykumar TN, Pomeranz I. Transient-fault recovery for chip multiprocessors. SIGARCH Comput Arch News 2003;31(2):98–109.
[13] Gong C, Melhem R, Gupta R. Loop transformations for fault detection in regular loops on massively parallel systems. IEEE Trans Parall Distrib Syst 1996;7(12):1238–49.
[14] Grama A, Gupta A, Karypis G, Kumar V. Introduction to parallel computing. Essex, England: Pearson Education Limited; 2003.
[15] Heidergott W. SEU tolerant device, circuit and processor design. In: Proceedings of the 42nd annual conference on design automation; 2005. p. 5–10.
[16] ITRS. International technology roadmap for semiconductors: http://www.itrs.net/links/2004update/2004update.htm; 2004.
[17] Karri R, Iyer B. Introspection: a register transfer level technique for cocurrent error detection and diagnosis in data dominated designs. ACM Trans Des Autom Electron Syst 2001;6(4):501–15.
[18] Lee C, Potkonjak M, Mangione-Smith WH. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: International symposium on microarchitecture; 1997. p. 330–5.
[19] Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hllberg G, Hgberg J, et al. Simics: a full system simulation platform. IEEE Comput 2002;35(2):50–8.
[20] Mahmood A, McCluskey EJ. Concurrent error detection using watchdog processors – a survey. IEEE Trans Comput 1988;37(2):160–74.
[21] Miskov-Zivanov N, Marculescu D. Soft error rate analysis for sequential circuits. In: DATE '07: proceedings of the conference on design, automation and test in Europe; 2007. p. 1436–41.
[22] Narayanan SHK, Son SW, Kandemir M, Li F. Using loop invariants to fight soft errors in data caches. In: Asia and South Pacific design automation conference, Shanghai, China, January 18–21; 2005. p. 1317–20.
[23] Ng TP. Replicated transactions. In: Proceedings of the 9th international conference on distributed computing systems; 1988. p. 474–81.
[24] Rashid MW, Tan EJ, Huang MC, Albonesi DH. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In: IEEE PACT; 2005. p. 315–28.

[25] Ray J, Hoe JC, B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture; 2001. p. 214–24.
[26] Reinhardt SK, Mukherjee SS. Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th annual international symposium on computer architecture; 2000. p. 25–36.
[27] Rotenberg E. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In: Proceedings of the 29th annual international symposium on fault-tolerant computing; 1999. p. 84.
[28] Somani AK, Tridandapani S, Sandadi UR. Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location. IEEE Trans Comput 1995;44(7):865–77.
[29] Sundaramoorthy K, Purser Z, Rotenburg E. Slipstream processors: improving both performance and fault tolerance. In: Proceedings of the 9th international conference on architectural support for programming languages and operating systems; 2000. p. 257–68.
[30] Tamir Y, Tremblay M. High-performance fault-tolerant vlsi systems using micro rollback. IEEE Trans Comput 1990;39(4):548–54.
[31] Unsal OS, Koren I, Krishna CM. Towards energy-aware software-based fault tolerance in real-time systems. In: Proceedings of the international symposium on low power electronics and design; 2002. p. 124–9.
[32] Vijaykumar TN, Pomeranz I, Cheng K. Transient-fault recovery using simultaneous multithreading. In: Proceedings of the 29th annual international symposium on computer architecture; 2002. p. 87–98.
[33] Wang F, Xie Y, Rajaraman R, Vaidyanathan B. Soft error rate analysis for combinational logic using an accurate electrical masking model. In: VLSID '07: proceedings of the 20th international conference on VLSI design held jointly with 6th international conference; 2007. p. 165–70.
[34] Wilton S, Jouppi N. CACTI: an enhanced cache access and cycle time model. IEEE J Solid-State Circ 1996;31(5):677–88.
[35] Zhang W. Enhancing data cache reliability by the addition of a small fully-associative replication cache. In: Proceedings of the 18th annual international conference on supercomputing; 2004. p. 12–9.
[36] Zhu D, Aydin H. Reliability effects of process and thread redundancy on chip multiprocessors. In: DSN'06: proceedings of the DSN 2006 workshop on architecting dependable systems; 2006.