An Adaptive Approach for Online Fault Management in Many-Core Architectures

Cristiana Bolchini, Antonio Miele, Donatella Sciuto
Dip. Elettronica e Informazione - Politecnico di Milano
P.zza L. da Vinci, 32 - I20133 Milano - Italy
{bolchini|miele|sciuto}@elet.polimi.it

Abstract—**This paper presents a dynamic scheduling solution to achieve fault tolerance in many-core architectures. Triple Modular Redundancy is applied on the multi-threaded application to dynamically mitigate the effects of both permanent and transient faults, and to identify and isolate damaged units. The approach targets the best performance, while balancing the use of the healthy resources to limit wear-out and aging effects, which cause permanent damages. Experimental results on synthetic case studies are reported, to validate the ability to tolerate faults while optimizing performance and resource usage.**

I. Introduction

We are witnessing trends in technology, fabrication processes and computing architectures that lead to the design and development of multi-core and many-core systems constituted by a relevant number of relatively low-cost execution resources (e.g., processors and configurable accelerator units) to achieve high-performance while leveraging on energy consumption. These trends must cope with increasingly unreliable devices, affected by the shrinking of components' size, variations in the manufacturing process and increased transient errors caused by radiations and noise fluctuations. We aim at defining an approach for the implementation of a many-core architecture able to cope with both permanent and transient faults to guarantee a correct execution of the running applications while optimizing performance. We propose an *adaptive fault management* mechanism acting at the scheduling level, whose main task is to monitor the architecture's processing units (PUs) and to exploit only the healthy ones. In this paper, we envision a working scenario based on a many-core architecture, such as the Platform 2012 many-core accelerator (P2012, from a cooperation between STMicroelectronics and CEA, [1]), for the execution of multi-threaded applications. The architecture consists of a configurable fabric constituted by several clusters, each one aggregating a multi-core processing engine. The class of applications that best exploits this architecture includes multimedia (video/audio) applications, typically executing few tasks on a large amount of data. Our proposal introduces reliability properties by acting at two different levels; at the *fabric* level, where the application is dispatched to the available clusters, as well as locally, in the *cluster*, where the processors and hardware accelerators are exploited to execute the application threads.

This work is partially supported by EU-ARTEMIS SMECY project, grant no. 100230.

The proposed solution introduces a reliability-aware resource management layer, whose main goal is apply software-based techniques for detecting and tolerating faults occurring in the processing elements, and to dispatch application threads to the healthy resources only. The layer maintains information on the status of health of the architecture resources, while adopting a strategy to balance the load to limit negative bias temperature instability (NBTI), which leads to degradation effects [2].

The rest of the paper is organized as follows. Section II presents the background. Then, Section III discusses the proposed approach in all aspects, while Section IV reports the results of its application to a set of case studies, to evaluate the discussed approach. Section V discusses the related work comparing our proposal against past approaches. Final considerations and future work are discussed in the closing section.

II. Background

In this work we refer to the P2012 architecture, in development, however the proposed approach is general in its assumptions and can be adopted also for other architectural solutions.

Architecture model. P2012 [1] is a many-core computing fabric, highly modular and configurable; as shown in Fig 1a, it is hierarchically organized in multiple *clusters* of computational units, connected by means of an asynchronous Network-on-Chip (NoC). A *fabric controller* (FC) serves as the control interface between the platform and the connected computing system, and as a dispatcher of the applications. The controller can be programmed to assign with a specified off-line or on-line policy each application to a *single* cluster.

Each cluster is a multiprocessor system featuring up to 16 tightly-coupled processing units (PUs) sharing multi-banked instruction and data memories through a local NoC. The cluster also contains i) a *hardware synchronizer*, for the management of parallel applications (thread creation, their scheduling on the processing units and synchronization with barriers), and ii) a *cluster controller* (CC), devoted to the communications with rest of the platform and the booting of the application.

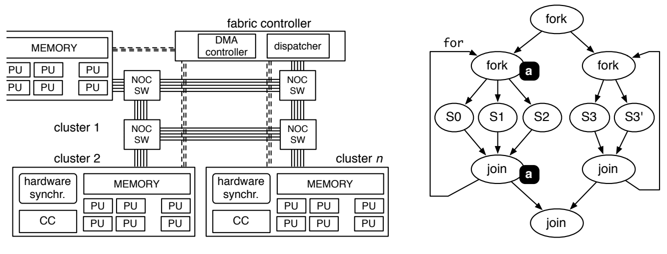

Figure 1. Reference a) architecture and b) programming models.

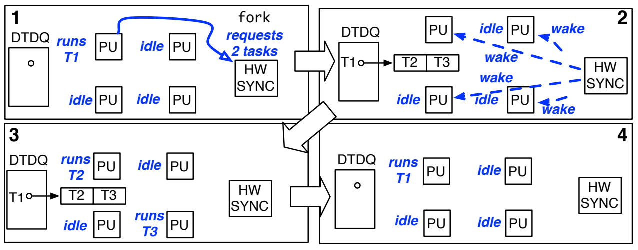

Figure 2. Application execution.

Application model. The programming model usually adopted for implementing parallel multi-threaded applications for many-core architectures is the `fork-join` model (for instance implemented by Open-MP paradigm [3]). The application starts with a main thread, and generates a set of new children threads for the execution of parallel independent elaborations; moreover, in turn, each one of the children threads may request a `fork`. Thus, each thread that performs a `fork` waits for all children threads to join before continuing the execution. Fig. 1b shows a graphical representation of the fork-join model. In particular, two separate nodes, that we also call *tasks*, are shown for a single thread to depict the `fork` and `join` instructions, however these pairing nodes (labeled **a** in Fig. 1b) constitute a single thread.

Application execution. The application is executed by a single cluster as follows (see [4] for details). The CC assigns the execution of the main thread to one of the PUs, while the others are in idle mode. When a `fork` instruction is met, the PU requests the creation of the children threads to the hardware synchronizer. This module adds a new entry containing the list of descriptors of the created threads into a specific table, called Dynamic Task Dispatching Queue (DTDQ), and sends an event to wake all the idle PUs. Each one of them iterates through the DTDQ table to execute all the ready threads, returning to an idle state when no ready threads exist. When the entire list of forked threads is executed, the entry is removed from the DTDQ and the parent thread is resumed on the same PU where it had been blocked (see an example in Fig. 2). At the end, the CC transmits to the FC the application outputs.

Fault model. The *single failure model* is adopted; one PU exhibits (transiently or permanently) an incorrect behavior by either producing erroneous results in the computation or by stopping responding (silent failure). Subsequent transient faults in different units are managed, provided the time between two consecutive events allows for the detection of such an erroneous condition. It is possible to remove the limitation of the single failure by leveraging on redundancy to tolerate also multiple concurrent errors.

III. Dynamic Fault Management

The proposed dynamic fault tolerant approach introduces a two-levels system to enforce fault management policies: two additional fault management layers are introduced in the architecture, one in the FC (centralized, at top level), the other in CC (distributed, at a lower level). Each one of these layers can be exploited or not, depending on the expressed reliability requirements; in particular, for each application a reliable or performance-oriented execution mode can be specified; thus, while in the former case fault-tolerant strategies are employed, in the latter case, computation is carried out in a nominal way, with no performance penalty. In this first proposal, we focused on the cluster layer; by acting at this level, the method achieves a prompt identification of failures in the PUs, to limit the risk of fault accumulation and masking. Future work will be devoted to a refinement of its policies, as the architecture becomes more mature.

A. Cluster controller's fault management layer

The CC is in charge of guaranteeing the correctness of the computations of the executed applications, and of identifying the occurrence of permanent failures. For this purpose, an additional layer has been added, cooperating with an enhanced version of the existing modules.

A few assumptions have been adopted. We assume the CC and the hardware synchronizer to be designed as hardened at the architectural level. This is necessary to avoid that a fault affecting their activity would not be detected and would cause the failure of the overall cluster execution without any alert (single point of failure). The cluster is provided with a memory protection mechanism allowing each thread to write only its private variables, input data and output data. This protection system prevents a thread affected by a fault from randomly corrupting the memory of the other threads running on the other cluster's processing units, leading to the failure of several threads (as in [5]). Finally, we will exploit the existing watchdog timers for detecting violations of execution deadlines caused by faults. Therefore, we can assume that any physical fault causes the failure of the single PU running a thread which will produce a wrong result within the expected deadline. The various aspects of the fault management layer are discussed in the following sections.

1) Fault mitigation strategy: The fault mitigation strategy we propose is based on redundant executions of the application. In particular, to tolerating the single functional failure of a PU, we triplicate the application (i.e., Triple Modular Redundancy – TMR). More precisely, we propose a *loosely-coupled application-level TMR* schema: when receiving a new application, the CC generates three replicas of the main thread; creating three identical, independent, execution flows of the same application, on the same cluster. We added in the fault management layer a *replica table*, for tracing the triplets of the generated replicas. The table is updated on thread creation and corresponding replicas are identified by means of thread IDs. Once a replica triplet is executed a voter thread is issued for output comparison. Do note that TMR is applied only when required by the user; otherwise the application is executed as discussed in Sec. II.

2) Adaptive scheduling: The scheduling and the execution of the ready threads (Sec. II) has been extended to manage the reliability requirements. First, replicas belonging to the same triplet have to be scheduled on different PUs. Therefore, the replica table is also used for storing information on the unit each thread is allocated on, thus, the hardware synchronizer will avoid that the same processing unit attempts to execute more than one of the same replicas in the DTDQ. Should a memory violation be signaled or a watchdog timeout expire during the execution of a thread, the replica table is updated with the information about the failure. Furthermore, when a thread completes the execution, it is tagged as *executed* in the replica table, but it remains blocked on a barrier preventing the joining of the parent thread until the voting is performed. When all the three replicas finish, the CC executes the voting task, and, finally, joins the parent threads. The voting task is executed by the CC since it is the only fault tolerant unit in the cluster, guaranteeing the correctness of the voting.

Indeed, the loosely-coupled approach is able to better adapt to the cluster architecture than a fully connected one. In fact, the latter would require the partitioning of all available resources into groups of three elements, to concurrently execute replicated threads (e.g., [6]). However, seldom the number of resources is exactly a multiple of three, and, most important, the availability of the resources is actually expected to change at runtime, due to failures and resource usage.

3) Faulty unit diagnosis and management: The fault management layer is also in charge of classifying faults as transient/permanent and of adopting the appropriate recovery actions. The layer uses two support tables, a resource health table and an error log table. The *local resource health table* (LHT) stores information on the status of cluster and its PUs. Each processing unit may assume three different states: i) *healthy* if the PU is operational, ii) *damaged* if a permanent failure has been identified, iii) *under-analysis* if the PU is suspected to be damaged, requiring specific diagnostic procedures to be executed.

The *error log table* stores the history on the errors occurred in the computation, to determine whether the fault is transient or permanent (on the basis of its occurrence), exploited also during the diagnosis process. The log is updated by the CC if a mismatch is detected during the execution of the voter thread. The table is analyzed to determine whether the failure may be due to a permanent or a transient fault, based on errors frequency (by using classification strategies similar to [7]). Suspected PUs are put in *quarantine* by setting its status to *under-analysis* in LHT. In this way, the PU becomes unavailable for the scheduler, and a diagnosis thread is started on that resource. If the result of the diagnosis task is a permanent failure, the PU is tagged has *damaged* and will not be used; otherwise, the resource status is restored to *healthy* and the unit will be again available to the scheduler.

The fault management layer is provided also with additional scheduling policies to prevent and limit wear-out and aging effects (e.g. NBTI). In particular, the LHT contains for each PU its utilization level. Thus, to balance the average workload of each unit, when there are new ready threads in the DTDQ, the hardware synchronizer will awake the PUs according to this new index.

B. Fabric controller's fault management layer

The FC has also been enhanced to perform a reliability-aware dispatching of the applications to be executed. We here present a brief description of the features of the new layer. The FC has a *global resource health table* (GHT) that summarizes the status of the overall architecture. The table contains a line for each cluster specifying the i) the busy/idle status, ii) the percentage of damaged resources, and iii) an utilization index (similar to the one used in the cluster). On top of this, we define a basic dispatching strategy aimed at limiting wear-out and aging effects, by allocating the application to be executed *to the first idle cluster, with the minimum utilization index*. Moreover, since the performance achievable by a cluster are affected by the ratio of its healthy resources, the FC keeps track of the health status of each cluster (in the GHT) and when the number of healthy resources drops below a fixed threshold, it disables the entire cluster.

IV. Experimental Results

We evaluate the performance penalty introduced by the fault tolerance layer by comparing it against the classical *tightly-coupled TMR* solution (with PUs partitioned into triplets). Our experiments are performed using a SystemC ([8]) transaction-level simulator of the P2012 architecture with a 16-cores cluster. We considered a large set of synthetic application `fork-join` graphs, consisting of 10, 20, 30 and 40 threads, respectively. Each node of the graph has been annotated with a latency ranging from 500ns to 5000ns, while voter threads require 500ns. We evaluated performances in a fault-free situation, and after the occurrence of faults, disabling 1, 2 and 3 PUs, respectively.

The graphs in Fig. 3 report the result of the comparison; the execution times are normalized w.r.t. the execution of the application without fault tolerant mechanisms. Results show that in the faulty-free scenario, both solutions present an overhead with respect to the normal execution, ranging from 50% to 100%, for smaller to bigger applications. As expected, the availability of numerous PUs limits the inherent penalty introduced by the TMR method. Note that, in the fault-free situation, the *tightly-coupled TMR* technique outperforms our proposal in about half of the experiments; the scheduler better fits the triplicated threads on the regular groups of PUs triplets, with respect to the situation of 16 independently available cores. This anomaly will be taken into account in future work, to try to exploit such element to improve performance. As

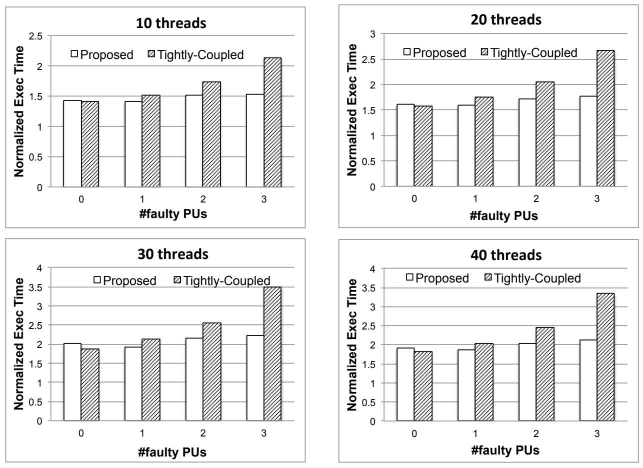

Figure 3. Fault-Tolerance related performance overheads.

expected, the traditional solution provides worse performance when permanent faults occur, with performance penalty overheads ranging from 7-11% (a single faulty PU) to 40-57% (3 faulty PUs), quite uniformly w.r.t. the application sizes. Lower values refer to the small applications with only 10 threads, easily accommodated on the available 16 cores; for all other application sizes, improvements are basically similar, as the number of cores is small with respect to the existing threads and the loss of a unit has a relevant impact on the overall performance.

Finally, we also analyzed the efficiency of the adopted load distribution strategy vs. the traditional "first available resource" scheduling, using an *utilization index*. Results show an average 19% decrease of the maximum resources busy-time, a significant step towards reducing possible wear-out situations.

V. Related Work

A comprehensive survey and comparison of approaches for on-line detection and recovery for multicore processors is presented in [9], discussing, among the others, solutions based on redundant execution to achieve fault detection/tolerance. Most of these works ([10], [6]) present interesting solutions based on code replication and eventually checkpointing, requiring though architectural support to manage the replicated threads and the comparison/voting activity, such that reliability would always be enabled. Moreover, most of these approaches usually considers a single threaded application [11]. In our proposal, we offer a flexible solution, allowing multi-threaded applications to be executed in a fault tolerant fashion, or not, based on the user's requirements, since the architecture is dynamically exploited to achieve reliability, when required. A similar feature is offered by the work proposed in [5], although the approach is mainly based on a architectural solution consisting of loosely-coupled processors. TriThread [12] is another application-level approach where OpenMP programs are hardened with redundant threads; differently from us, hardening is performed at design time, and the replicated code is dynamically scheduled on the different cores, with no specific heuristic.

Another class of available proposals refers to static scheduling strategies suitable when executing a set of predefined applications (e.g., [13], [14]). In particular, application hardening and scheduling are determined at design time, by taking into account the different possible situations that may arise at run time. The approach is affected by the complexity of analyzing all possible faulty scenarios and by only virtually acting in an adaptive way; moreover, only transient faults are considered.

VI. Conclusions and Future Work

This paper has presented a new adaptive application-level approach for achieving fault tolerance in many-core architectures. The proposed solution exploits thread redundancy to tolerate errors due to both transient and permanent faults. Application threads are dynamically scheduled on the healthy units of the architecture, to keep high performance, and to minimize device degradation. Future work is devoted to enhance the dynamic management of the resources in the fabric controller.

References

[1] STMicroelectronics and CEA, "Platform 2012: A many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology," in *Research Workshop on STMicroelectronics Platform 2012*, 2010.

[2] J. Sun, R. L. Lysecky, K. Shankar, A. K. Kodi, A. Louri, and J. M. Wang, "Workload capacity considering NBTI degradation in multi-core systems," in *Proc. ASP-DAC*, 2010, pp. 450–455.

[3] (2011, Sept.) The OpenMP API specification for parallel programming. [Online]. Available: http://openmp.org/wp/

[4] M. Ojail, R. David, K. B., Y. Lhuillier, and L. Benini, "Synchronous Reactive Fine Grain Tasks Management for Homogeneous Many-Core Architectures," in *Proc. Workshop on Parallel Progr. Run-Time Mgmt Techniques for Many-Core Architectures*, 2011, pp. 144–150.

[5] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Mixed-mode multicore reliability," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 169–180.

[6] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *Proc. Conf. Dependable Systems and Networks*, 2007, pp. 317–326.

[7] C. Bolchini and C. Sandionigi, "Fault Classification for SRAM-Based FPGAs in the Space Environment for Fault Mitigation," *Embedded Systems Letters*, vol. 2, no. 4, pp. 107–110, Dec. 2010.

[8] Open SystemC Initiative, "www.systemc.org."

[9] D. Gizopoulos et al., "Architectures for Online Error Detection and Recovery in Multicore Processors," in *Proc. Design, Automation Test Europe Conf.*, 2011, pp. 1–6.

[10] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proc. Intl Symp. Computer Architecture*, 2002, pp. 99–110.

[11] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proc. ACM Int. Symposium on Computer Architecture*, 2007, pp. 470–481.

[12] H. Fu and Y. Ding, "Using Redundant Threads for Fault Tolerance of OpenMP Programs," in *Proc. Int. Conf. Information Science and Applications*, 2010, pp. 1–8.

[13] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems," in *Proc. ACM Conf. Design, Automation and Test in Europe*, 2006, pp. 706–711.

[14] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware Co-synthesis for Embedded Systems," *J. VLSI Signal Process. Syst.*, vol. 49, pp. 87–99, 2007.