A Context Saving Fault Tolerant Approach for a Shared Memory Many-Core Architecture

Eduardo Wächter*†, Nicolas Ventroux†, Fernando G. Moraes*

* FACIN - PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
†CEA, LIST, Embedded Computing Laboratory – 91191 Gif-sur-Yvette CEDEX – France
nicolas.ventroux@cea.fr, {fernando.moraes, eduardo.wachter}@pucrs.br

***Abstract*—Mechanisms for runtime fault-tolerance in many-core architectures are mandatory to cope with transient and permanent faults. This issue is even more relevant with aggressive technology nodes due to process variability, aging effects, and susceptibility to upsets, among other factors. This work proposes to save periodically the context and to re-schedule tasks to the last reliable known state and avoid the faulty processor. This technique is implemented on an embedded multicore architecture named P2012. The proposed fault-tolerant approach induces a limited overhead of 9.37% in an industrial image processing application while guaranteeing a full-error recovery if any error is detected.**

Keywords—NoC-based MPSoC; fault recovery; context saving; checkpointing; rollback

I. INTRODUCTION AND RELATED WORKS

The probability of transient (e.g. crosstalk, SEUs) or permanent faults occurrence due to manufacturing errors [1] and wearout (e.g. electromigration) [2] is increasing with the technology scaling. In this scenario, a many-core architecture that does not take into account faulty processors can no longer guarantee the correctness of its behavior during its lifetime. In addition, if there is no Fault Tolerance (FT) management of healthy and faulty processors, the whole system may be blocked or subject to erroneous computations, leading to a malfunction of the system. For this reason, it is crucial that the system can self-adapt itself in order to isolate a faulty processing element (PE) and recover from a reliable previous context.

The Authors in [3] present ReVive: a checkpoint/rollback mechanism for architectures with processors, caches and memory interconnected by an off-chip network. They implemented a *partial separation with logging* checkpoint mechanism. This approach proposes a partial separation, where checkpoint data and working data are one and the same, except for those elements that have been modified since the last checkpoint. This kind of approach requires the memory to be divided into pages, with a hardware directory controller responsible for the access to the memory. The results on a 16-processor system indicate that the average error-free execution time overhead of using ReVive is only 6.3%.

In [4] the authors propose a Chip-level Redundant Threading (CRT) to detect transient faults on Chip Multiprocessors (CMPs). The approach is to execute two copies of a given program on distinct cores and then compare the stored data. CRTR (CRT with Recovery) achieves fault recovery by comparing the result of every instruction before commit. Once detecting different results, the microprocessor could be recovered by re-executing from the wrong instruction. The results showed that the performance overhead of the context saving when compared to the baseline processor is approximately 30%.

The *Reli* technique [5] proposes to change the micro-operations of instructions, which stores registers and data memory. They adopted two stacks used for storing the registers in the register file and for storing the data memory values that changed. Results showed an overhead of 1.45% in the execution time on a faulty-free scenario and incurs area overhead of 45% on average.

The *DeSyRe* project [6] presents an MPSoC framework for FT purposes. As error recovery technique, they propose the checkpoint and task re-execution for an MPSoC with seven cores. The Authors do no evaluate the checkpoint technique. For this reason, there are no results related to the overhead in a fault-free scenario. However, the evaluation of the application re-execution for a matrix multiplication application in a scenario with 20% of tasks being faulty, the execution time doubles.

Gizopoulos et al. [7] classifies error recovery techniques into two categories: forward error recovery (FER) and backward error recovery (BER). FER techniques detect and correct the errors without requiring to rollback to a previous correct state (e. g. using Triple Module Redundancy - TMR). The BER techniques periodically save (checkpoint) the system state and rollback to the latest validated checkpoint when a fault is detected.

Some of the methods shown an overhead without faults smaller than 20%, considered an acceptable overhead [3][5]. However, these approaches target distributed systems [3] or require modification in the ISA and dedicated hardware [5]. Other methods present a larger overhead in the presence of faults [6] or require redundant executions, wasting processing resources [4].

The *goal* of the present work is to propose a lightweight error recovery technique for multi-core systems, targeting the P2012 multicore platform. In this paper, we propose to add a fault-tolerant feature to the P2012 architecture by using an automatic checkpointing and recovering method. If a fault is detected, the previously saved context is restored, allowing the system to continue its execution with unaltered data.

Contributions. The contributions of this paper are: (1) a checkpointing/recovery method implementation in the P2012 architecture and (2) an isolation technique to isolate a faulty internal core.

Two important *assumptions* adopted in the current work: (1) fault detection is out of the scope of this research, implementations have already been proposed, such as in [6]; (2) there are no pragmas or code added by the software designer, allowing context saving at any moment of the application execution.

978-1-4799-8391-9/15/$31.00 ©2015 IEEE

II. FAULT-TOLERANT REFERENCE PLATFORM

The P2012 multicore architecture is an area- and power-efficient many-core architecture for next-generation data-intensive embedded applications such as multi-modal sensor fusion, image understanding, or mobile augmented reality [8]. The P2012 contains multiple processor clusters implemented with independent power and clock domains, enabling fine-grained power, reliability, and variability management. P2012 can reach 19 GOPS (with full floating point support) in 3.8mm^2 of silicon with 0.5 W power consumption.

A. Architecture

Figure 1 presents an overview of the P2012 architecture. P2012 is a GALS fabric of tiles, called *clusters*, connected through an asynchronous global NoC (GANOC) [9]. Each cluster has access to an L2-shared memory and to an external L3-shared memory. Each P2012 cluster aggregates a multi-core computing engine called *ENCore* and a cluster controller (CC). The *ENCore* contains 16 STxP70-V4 processing elements (PEs). Each PE is a 32-bit load/store architecture with a 7-stage pipeline able to execute up to two instructions per clock cycle (dual issue), with a floating-point unit extension. Each core has an L1-16KB-private instruction cache and can share an L1-256KB tightly coupled data memory distributed in 32 banks (TCDM) with the other cores of its cluster through a logarithmic interconnect.

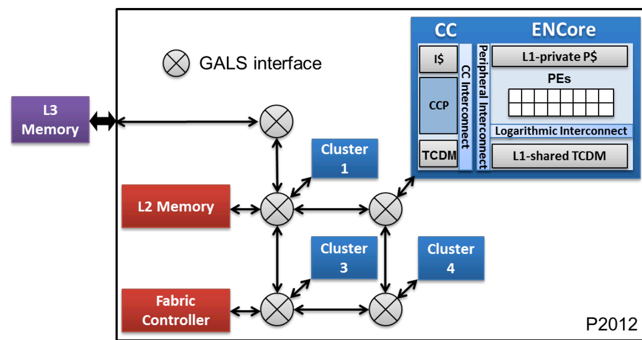

Figure 1 – P2012 architecture.

The CC is the manager processor of its cluster. The CC encloses a processor named CCP, a DMA subsystem and three interfaces: one to the ENCore, one to the GANOC and one to the local peripheral network to plug hardwired accelerators. The CC processor also adopts the STxP70-V4 processor, 16-KB of program cache and 32-KB of local data memory. The cluster controller processor, together with its peripherals, is in charge of booting and initializing the *ENCore*. It also performs application deployment on the *ENCore*. The DMA sub-system has two independent DMA channels. It performs the data block transfers from the external memory to the internal memory and vice-versa while the various cores are operating. The CC interconnects supports intra and inter-cluster communication.

B. Software Stack

The software stack is named HARS [10] and it is based on a hardware-assisted runtime software. It is composed of resource management features, multiple execution engines to support different programming models, and synchronization primitives relying on a hardware module named HardWare Synchronizer (HWS).

The HWS is dedicated to accelerate synchronization primitives on massively parallel embedded architectures. It is designed as a peripheral to be integrated into architectures using load/store operations, providing their runtime software with efficient synchronization implementations even for architectures without atomic operations support. It can also remove polling issues related to spin-lock operations. A specific set of software synchronization primitives based on this hardware accelerator can be used by the different cores to perform synchronizations. Thus, instead of using a software instruction requiring an atomic memory read/write access, the synchronization primitives leverage the HWS atomic counters to implement locking. Moreover, the runtime software uses sleep locks to put the processor in a waiting state until it is awakened when the resource is free.

HARS proposes a small set of execution engines covering a wide range of parallel programming styles. Two main execution engines are implemented: conventional multi-threading for coarse grain parallel expression (suitable for thread-level or task-level parallelism) and synchronous and asynchronous reactive tasks management for fine-grain parallelism (suitable for data-level parallelism).

Finally, an API enables the software designer to have access to all communication primitives, parallel task execution triggering and control of the synchronization features presented in other layers.

C. Execution model

In this paper, a conventional multi-threading execution model based on fork-join mechanisms has been chosen. The PE that executes the fork is referred as *master PE* (PE_m). Any of the 16 PEs of the *ENCore* may be select as PE_m. As showed in Figure 2, PE_m executes the sequential part of the application and can delegate tasks to other processors, parallelizing the execution.

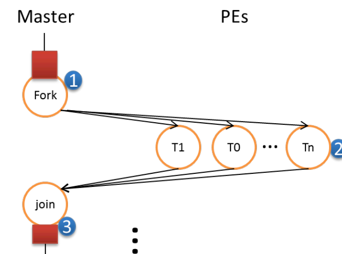

Figure 2 – Execution Model in P2012 with HARS. (1) master forks parallel tasks, (2) other PEs execute the tasks, and (3) the master does the join.

PEs only executes tasks that were forked by PE_m. To execute the fork, the PE_m populates a table with tasks to be executed. The fork procedure loads the local shared memory within the cluster with the data and instructions to be executed by the parallel tasks. After the load procedure is made, each PE executes the tasks. When there are no more tasks to be scheduled, the PE_m waits until all tasks have finished their execution to join the tasks. Every PE that is not doing a fork operation executes a scheduling loop. This loop searches for jobs to be executed by scheduling ready tasks from this table.

Each PE accesses a dedicated shared memory space, which is released when the task finishes its execution. At this point, the local memory in the cluster accessed by the PE has no useful information about the execution on PE_m and can be discarded. The fork-join process can be repeatedly executed, but the PE_m must wait all tasks to finish their job before the join.

III. FAULT TOLERANT EXECUTION MODEL PROPOSAL

As stated in the Introduction, this paper proposes a fault tolerant approach to tackle faults occurring in the processors.

A. Context Saving and Restoring

According to our execution model, only the context of PE_m is saved, as well as the global shared memory space. Thus, the context saving/restoring process is performed before the fork and after the join. This guarantees a coherent state for all PEs and eases the management of faults.

Thus, the execution context that must be considered is a structure composed by the 32 PE_m's registers, the *.data* section that stores all the shared uninitialized data, the *.bss* section that stores all the shared initialized data, and the PE_m stack which is locally stored in the CC L1-data memory.

This structure is stored in the L3 memory, accessible by all the clusters. All accesses to this memory are made through the GANoC, inducing network traffic. The access time is higher when compared to the local shared memory within the cluster. The structure is allocated at runtime according to the size the application needs.

B. Task interruption and faulty PE isolation

The HARS software stack in P2012 does not allow the PEs to send an interruption to the PE_m. Then, it is not possible to interrupt the fork execution at the exact moment the fault is detected. Our proposal is to use an atomic counter to store the information if the PE is faulty or not. At the end of the parallel task execution, PE_m verify if there was an error in some PE, reading its atomic counter. If a given PE is faulty, it is isolated from the execution processor list and consequently will not execute any other task. Then, PE_m starts the recover context procedure.

C. Fault-tolerant mechanism

Figure 3 presents the proposal of the Fault Tolerant (FT) execution model. Before the sequential execution is forked, the master saves the application context. This means that it stores in the global shared memory (in this order): (1) all processor registers; (2) its stack: (3) its *.bss* section and (4) its *.data* section. At the end of the fork/join process, the master checks if any of the PEs were hit by a fault and if needed, it triggers the recover context procedure. If there was no fault, the execution continues normally.

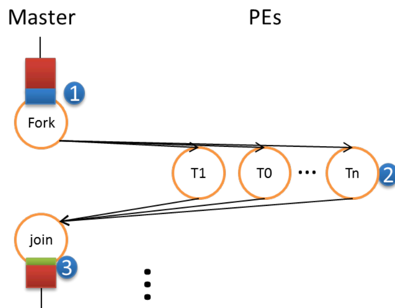

Figure 3 – Fault Tolerant Execution Model: In (1) the master executes a context saving and in (3) it verify if there was a fault, if positive, the context is restored and the fork is re-executed avoiding the faulty PE.

IV. EVALUATION AND RESULTS

This section evaluates the overhead induced by the FT proposal. All scenarios are executed in the SDK of P2012 released by ST Microelectronics. For the results, only 1 cluster is considered, and all the communications between tasks are made through the global shared memory space in the L3 and memory accesses are made through the GANOC.

A. Applications description

The first application is synthetic, with it task graph presented in Figure 4. A parameterizable number of NOP instructions (***N***) define the task size. It is also possible to parameterize the number of task (***T***), and the number of iterations (***R***).

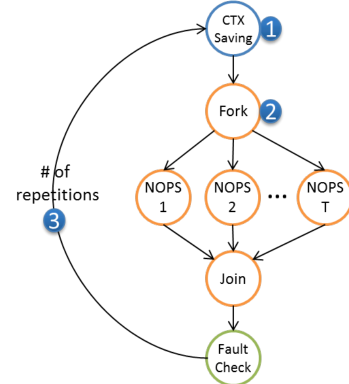

Figure 4 – Task Graph of the synthetic application. (1) The PE_m executes the context saving, (2) the fork splits the execution in *T* tasks, each one executing *N* number of *NOP* instructions, and (3) this process is replicated *R* times.

The second application is an industrial application named Human Body Detection and Counting (HBDC). It consists in processing an image sequence to determine the background image and subsequently the moving objects of the scene. The first phase uses the Mixture of Gaussian (MoG) technique [11]. It is forked in 60 tasks, each one taking around 340,000 clock cycles to execute. Then, the remaining tasks are sequential. The moving objects are classified to determine whether they correspond to human shapes. 64 image frames are processed.

B. Evaluation of the method with Synthetic Application

Figure 5-a measures the impact of the context saving varying ***N***. The time to save the context is not a function of the task size (*N*). Thus, the size of the parallelized tasks should mask the context saving overhead. As shown in the Figure, the execution overhead reduces as *N* increases. A task with 10,000 *NOPs* has an overhead close to 20%, which is considered an acceptable overhead. The next experiments use N=10,000 as reference for the task size.

The next experiment evaluates the impact of the context saving, varying the size of the sections *.data* and *.bss* (Figure 5-b). The amount of data to save is the main limitation of the approach. A trade-off has to be defined between the tasks' execution time and the context data size. Then the programmer can choose an acceptable overhead cost of the context saving.

The context saving is disabled to enable the evaluation of the fork overhead. Figure 5-c shows the execution time overhead varying ***R***. Figures shows that the fork execution takes approximately 6% of the execution time. Since this overhead is independent of the number of repetitions, the fork/join process has a limited small impact in the performance.

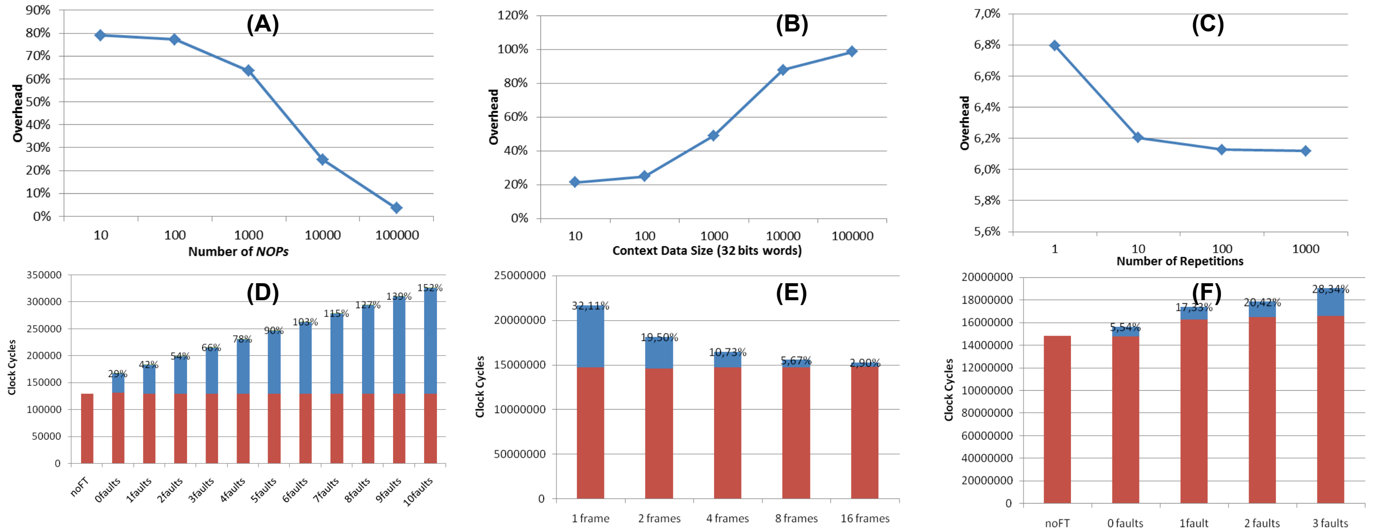

Figure 5 – (a) Execution time overhead varying the number of NOPs in each task (*T*=10, *R*=10). (b) Execution time overhead of context saving changing the context data size from 10 to 10k words of 32 bits (*T*=10, *R*=10, *N*=10,000). (c) Fork overhead varying the number of repetitions (*T*=10, N=10,000). (d) Application execution time overhead for scenarios with no context saving, and the overhead for scenarios where there is overhead increasing the number of faults (*T*=10, *R*=10, *N*=10,000). (e) Execution time overhead without faults when executing context saving from each frame to each 16 frames. The bars show the context saving overhead and the execution time. (f) Application execution time with no Context Saving, the overhead induced by the context saving and the overhead induced by the context saving plus the recovery time for one, two and three faults. The percentages represent the overhead compared to the baseline. The highlighted part represents the time executed saving the context.

Figure 5-d shows the execution time for the synthetic application without context saving (*noFT*), with the FT method and no injected fault (*0faults*), and with the FT method and a variable number of injected faults. The context saving implies a 29% overhead (N=10,000), and for each injected fault there is an increase of 12% for the context restoring and fork rescheduling.

C. Evaluation of the method with HBDC Application

Figure 5-e shows the execution time overhead when context saving is executed according to a variable number of frames. Saving the context at each eight frames increases the execution time by 5.67%. This means that the background images will be restored as it was eight frames back if a fault is detected. For this application, the checkpointing frequency has only a QoS impact that will depend on the application frame rate. With a high frame rate, losing some frames will not affect the application, resulting in a good tradeoff between performance and quality.

Figure 5-f presents five executions of the HBDC application, assuming context saving at each eight frames. In the first column (*noFT*), there is no context saving, being the baseline execution time. The second column shows the overhead induced by the context saving with no fault insertion (9.37% compared to baseline). The last three columns show the overhead for one, two and three faults in different frames of the application. Note that the percentage represents the overhead compared to the baseline and the highlighted part represents only the context saving. As there are tasks to be re-executed, the task execution time increases when the number of faults grows.

V. CONCLUSION AND FUTURE WORKS

This work presented a Fault Tolerant Context Saving for a state-of-the-art shared-memory MPSoC. Results showed that the proposal was validated and could recover applications from faults occurring in PEs. Execution with an industrial application shows a good tradeoff between execution time overhead with no faults (5.54%) and with faults (17.33% - 28.34%). The proposal does not imply in hardware overhead or redundant executions, as works in the state-of-the-art.

Future works focus on two fronts. The first is to enable each parallel task to execute a context saving. The second is to implement a mechanism where only the modified segments of the shared memory would be replicated by the context saving.

ACKNOWLEDGMENTS

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, CAPES - project CAPES-COFECUB 708/11, and FAPERGS - project 2242-2551/14-8.

REFERENCES

[1] Borkar, S. Thousand Core Chips - A Technology Perspective. In: DAC, 2007, pp.746-749.

[2] Lienig, J. “Electromigration and its impact on physical design in future technologies”. In: ISPD, 2013, pp. 33–40.

[3] Prvulovic, M.; Zheng Zhang; Torrellas, J. “ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors”. In: ISCA, 2002, pp. 111–122.

[4] Gong, R.; Dai, K.; Wang, Z. “Transient Fault Recovery on Chip Multiprocessor based on Dual Core Redundancy and Context Saving”. In: ICYCS, 2008, pp. 148–153.

[5] Li, T.; Ragel, R.; Parameswaran, S. “Reli: hardware/software checkpoint and recovery scheme for embedded processors”. In: DATE, 2012, pp 875–880.

[6] Sourdis, I. et al. “DeSyRe: On-demand system reliability”. Microprocessors and Microsystems. 37, 8, 2013, pp 981–1001.

[7] Gizopoulos, D. et al. “Architectures for online error detection and recovery in multicore processors”. In: DATE, 2011, pp.1–6.

[8] Benini, L. et al. “P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator”. In: DATE, 2012, pp. 983–987.

[9] Thonnart, Y.; Vivet, P.; Clermidy, F. “A fully-asynchronous low-power framework for GALS NoC integration”. In: DATE 2010.

[10] Lhuillier Y. et al. “HARS: A hardware-assisted runtime software for embedded many-core architectures”. ACM Trans. Embedded Computing Systems v.13(3), 2014, 25 p.

[11] McLachlan, G. J.; Peel, D. “Finite mixture models”. New York: Wiley, 2000.