Fault Recovery Protocol for Distributed Memory MPSoCs

Francisco F. S. Barreto, Alexandre M. Amory, Fernando G. Moraes
FACIN - PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
francisco.barreto@acad.pucrs.br, {alexandre.amory, fernando.moraes}@pucrs.br

Abstract— **Fault handling mechanisms become more relevant as systems integrate more hardware logic. For instance, current multi-processor system-on-chips (MPSoCs) consists of hundreds of processors connected by an interconnection network. This type of system can only be cost effective if it can handle faults on its main components (i.e. processors and interconnect). Traditional fault recovery approaches for multi-processors were adapted from the domain of cluster of computers and might be more complex than required for common MPSoC applications domains. This paper presents a lightweight online fault recovery for embedded processors of MPSoCs based on distributed memory. This approach automatically restarts affected applications reallocating tasks to healthy processors. All steps are performed at the kernel level, without changing user application code. Results show very short recovery time, from 110 µs to 425 µs with a 100MHz clock, which are mostly dominated by the size of the reallocated tasks.**

Keywords— many-cores; MPSoCs; distributed memory; fault recovery; task migration

I. Introduction and Related Work

As systems become more complex, integrating more logic into a single chip including hundreds of embedded processors, methods for fault tolerance become mandatory. In the context of MPSoC and many-cores, there are two major classes of architectures: shared memory and distributed memory. Shared memory architectures are easier to program, but it might have scalability issues with a bottleneck at the shared memory. In this case, distributed memory architectures can be an alternative.

Most fault recovery proposals for many-cores assume the existence of a shared memory and multithreading programming [1][2][3], while other approaches adopt spare hardware [4][5] for redundancy. Task migration has been primarily used for performance purposes, to avoid hotspots and to perform load balancing [2][6]. Derin et al. [7] propose system level adaptive and fault tolerant techniques for NoC-MPSoCs, which uses dynamic remapping (task migration) of faulty PEs at execution time. This method does error detection and fault recovery with task migration where only the context of tasks is migrated. The code is not migrated because all PEs have the binary code of all tasks. There are also a large number of proposals on network fault recovery complementary to the proposed approach [8][9][10].

The *goal* of the present work is to propose a low-cost software-based fault recovery mechanism for faults on the processing elements (PEs) of an MPSoC with distributed memory architecture. The method detects faults at the PE, marks this faulty PE as unusable, temporally freezes the unaffected tasks of an application, automatically reallocates the affected tasks to healthy PEs, reconfigures the tasks addresses, and restarts the application.

The main *contribution* of this paper is a software-based method that automatically reallocates tasks of an application affected by faults at the PEs. The advantages of the method are: (*1*) low cost since no extra hardware, no spare PE, and no redundant software are required. Moreover, no application data context is saved, leading to a lightweight recovery strategy; (*2*) it automatically restarts the affected applications causing a short downtime which, for most dataflow applications, can be imperceptible by the user, thus, do not cause a system failure; (*3*) not limited to the shared memory parallel programming approach; (*4*) the fault recovery mechanism is decoupled from the test approach, enabling different test approaches according to the application's reliability and availability constraints; (*5*) the user application is not changed.

II. The MPSoC Model

The proposed method works on an MPSoC model described as a set of PEs interconnected by a given network topology as illustrated in Figure 1 and described as follows.

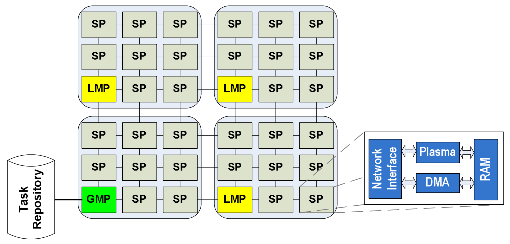

Figure 1 – MPSoC Model [12].

A. *The Architectural Model*

Each PE contains one processor (Plasma) with a private memory, a DMA controller, a Network Interface (NI), and a router. Moreover, the evaluated architecture is homogeneous, i.e., all PEs have the same architecture. However, the software part of the FT (Fault-Tolerant) approach is entirely written in C language. Thus, it can be compiled to a different target processor, supporting heterogeneous MPSoCs.

The network is supposed to be reliable, i.e. the messages are delivered even with the occurrence of faults. Fault tolerant NoCs such as [8][9][10] can be used. The experiments of this paper use mesh network topology only for the sake of simplicity. The proposed method is not influenced by the topology or any other network parameter.

The applications are modeled as task graphs, and multiple applications can be dynamically loaded via DMA, executing in parallel. The applications and their tasks are stored in an application repository (i.e. external memory), which is supposed to be reliable. A mapping function dynamically maps tasks [11] from the task repository onto PEs called *slave PEs (SP)*. A multi-tasking operating system (OS) enables to execute more than one task per *SP*.

The coordination of applications and OS-level services are executed by multiple PEs called *Local Manager Processors (LMP)*, improving system scalability and avoiding a single point of failure. There is a PE called *Global Manager Processor (GMP)* which coordinates the access to the task repository memory.

B. The Programming Model – Application Level

The communication model is message passing with distributed memories. At the application level, the communication API has two MPI-like primitives: a non-blocking *Send()* and a blocking *Receive()*. To implement a non-blocking *Send()*, a dedicated memory space in the OS, named *pipe*, stores each message written by tasks. Those messages are only sent to the network when requested by their respective target tasks. Incoming messages are transferred to the task memory space, being consumed when the task returns from idle OS state.

C. The Programming Model – OS Level

The OS provides services to the application layer. Those services include dynamic application mapping, load balancing, monitoring, QoS support, among others.

In the context of fault recovery, we use two previously existent OS services: (*i*) *task reallocated*, notifies the network address of a task to the other tasks of the same application; (*ii*) *task allocation:* it transfers the task binary code from the task repository to the local memory of a target PE.

III. Fault Recovery Method

This section initially justifies the use of task migration without saving data context and shows how this approach can be useful for certain applications. Next, it describes how the PE faults can be detected. Finally, the proposed fault recovery method is presented.

A. The Cost of Checkpoint and Rollback

A complete solution based on checkpoint and rollback has been used in the domain of cluster of computers [13] and has been recently adapted to shared-memory many-cores [1]. However, it has a series of drawbacks and overhead that might not be acceptable in the context of many-cores. These limitations include [13]:

– Checkpoint and rollback were primarily proposed for applications with long execution time (e.g. weeks). This is not the case for most common MPSoC applications, especially in the consumer electronics application domain.

– Most approaches require a very complex memory management for the context memory including, for instance, garbage collection to delete old contexts. Moreover, the memory usage for context scales with the system size, which is not suitable for MPSoCs.

– Most approaches require instrumenting the application code to perform a checkpoint, changing the user application.

– The periodic checkpoint process uses a considerable CPU power and network bandwidth to keep the distributed checkpoints consistency. Moreover, it makes the memory bandwidth issue even worse.

– Most checkpoint approaches do not handle asynchronous and non-deterministic events, such as interrupt requests.

On the other hand, current MPSoC applications are targeting primarily consumer electronic market, with ever increasing demand of processing power and multimedia capability. In this context, fault tolerance is not required to execute applications for weeks, as in the cluster domain. Fault tolerance is required to enable the system to work with high-end deep submicron process technologies that are affected by manufacturing, transient, and aging faults.

In this context, checkpoint and rollback might be overshooting, and its costs and drawbacks might be larger than acceptable. A lightweight approach can lose the data during the fault detection, but a fault recovery guarantees that the future data is processed. Despite the drawback of losing data, the proposed approach is much simpler than checkpoint, and rollback and it can still be useful for common applications in the MPSoC domain. For instance, a downtime of about 1 ms in a video processing application is acceptable. Other signal processing and dataflow applications can also benefit from the proposed fault recovery approach.

B. Testing the Processor Elements

Several test approaches can be used to test the processing elements [14]. We understand that the test is, at the same time, one of the most costly steps in a dependable method and it is mostly dependent on the application's reliability and availability constraints. This way, we devised a recovery method that can be used with different test approaches. Just to cite a few test methods:

(i) Periodic transparent BIST (implemented both in software or hardware) at the embedded local memory since this BIST approach does not delete the memory content [15];

(ii) Periodic software-based structural/functional self-test of the processor core [16];

(iii) Use of STUMPS-based logic BIST for internal bus, network interface, glue logic [15];

(iv) Dynamic verification and anomaly detection [14];

(v) Prognostic health monitors can be used to evaluate the PE's aging, switching on the proposed fault recovery protocol before a fault is detected [17].

The focus of this paper is not on the test, but on the fault recovery process. Consequently, we assume the existence a test mechanism that detects permanent faults at slaves PEs and this test mechanism fires the proposed recovery process.

C. The Proposed Application Recovery Protocol

The proposed application recovery protocol is implemented at the OS-level, in the *kernel* space, as services of the OS. The proposed protocol uses the two services mentioned before (II.C) and requires tree new services:

(i) *Fault notification message*: message sent from the fault detection mechanism to its local manager to inform the detection of a defective PE.

(ii) *Freeze task*: message sent from the local manager to a set of PEs so that they can put certain tasks in freeze state until the recovery is complete. A task in a freeze state cannot be scheduled by the OS and cannot send/receive messages, stopping the application affected by the fault. No context task data is saved in this step.

(iii) *Unfreeze task*: message sent from the local manager to a set of PEs, changing the task status from freeze to the ready state. The unfreezed task initially cleans up old data context, incoming messages, and starts up the task execution from the beginning. It works similarly to an 'application reboot' where any previous data is lost, however, the application is ready to process new data.

The protocol works as follows, with all actions executed in a manager PE (PE_M), LMP or GMP:

(1) PE_M receives a fault notification message, indicating a fault at PE_F.

(2) PE_M constructs a set $TF = \{t_1, t_2, ..., t_n\}$ representing the affected tasks running in the defective PE_F.

(3) PE_M identifies for each t_i in TF the applications affected the fault, constructing a new set TA = $\{A_1, A_2, ..., A_m\}$, being A_i an application to freeze.

(4) PE_M sends a *freeze message* for all tasks of each application in TA.

(5) PE_M remaps only the tasks in TF to healthy PEs. All other unaffected tasks stay on the same location. After the completion of the dynamic task mapping, the unaffected tasks receive the new addresses for the remapped tasks.

(6) PE_M sends the *reallocation* and the *unfreeze messages* for all tasks of each application in TA.

Figure 2 illustrates the pseudo-code executing at the manager kernel layer. Let capital symbols represent sets, and lower case symbols represent elements of a set. Let us assume the following functions: *getTasksFromPE* (line 1) receives the PE id and returns the set of tasks mapped to this PE; *getApp* (line 5) receives the task id and returns its corresponding app id; *getPE* (line 7) receives a task id and returns the PE where it is mapped; *dynamic_mapping* (line 11) receives a task and returns its PE address; setPE (line 12) assigns a task to a PE.

```
1.  TF=getTasksFromPE(PEf)
2.  TA= ∅
3.  // send freeze message
4.  for each t in TF
5.     TA= TA U getApp(t)
6.  for each app in TA
7.    for each pe in (getPE(app)-PEf)
8.      send(pe,freeze,app)
9.  // perform task reallocations
10. for each t in TF
11.   new_pe = dynamic_mapping(t)
12.   setPE(t,new_pe)
13.   send(new_pe,task_allocation,t)
14.   app = getApp(t)
15.   for each pe in getPE(app)
16.     send(pe,reallocated,new_pe)
17. // send unfreeze message
18. for each app in TA
19.   for each pe in getPE(app)
20.     send(pe,unfreeze,app)
```

Figure 2 – Fault recovery pseudo-code at the manager kernel.

TF (line 1) represents the set of tasks directly affected by the fault on PE_f. *TA* (line 5) represents the set of applications affected by the fault on PE_f. The code from line 7 to 8 searches for tasks that were not affected by fault and sends a *freeze* message to all of them. From lines 10 to 13, all the affected tasks are remapped to new PEs, transferring the task's binary code. At lines 14 to 16 the manager informs the new task locations and finally, lines 18 to 20 unfreezes all applications.

The *SP* pseudo code is illustrated in Figure 3. When the slave PE receives a freeze message, the task indicated in the message goes to the freeze state, and the kernel cannot schedule it until it is unfreezed. When the slave PE receives an unfreeze message, it performs 3 main actions: (*i*) clean the task's Task Control Block (TCB), returning the data to its initial state; (*ii*) assign the task to ready, so it can be scheduled by the kernel ; (*iii*) delete all pipe messages sent to the task, deleting any pending message.

```
msg = recv()
app_id = msg.app_id
switch (msg.service)
  case freeze:
    for each task in task_list
       if task.app_id == app_id
          task.state = freeze
  case unfreeze:
    for each task in task_list
       if task.app_id == app_id
         clean_task_tcb(task.id)
         task.state = ready
    for each pipe_slot in pipe
       if pipe_slot.app_id == app_id
          pipe_slot.status = empty
```

Figure 3 – Fault recovery pseudo-code at the slave kernel.

IV. EXPERIMENTAL SETUP AND RESULTS

All experiments were executed on an MPSoC with 16 PEs communicating through a wormhole best-effort NoC with a mesh topology and 16-bit channels. The system has a global clock of 100MHz. The maximum number of tasks per PE is adjusted from 1 to 5 tasks per PE. The application starts at the beginning of the simulation (cycle-accurate VHDL RTL model) containing from 4 to 12 tasks, depending on the experiment. A fault is injected at 2,500 µs in an SP. At this moment, a PE sends the fault notification to the LMP, triggering the recovery algorithm. The running application is a parallel sum of vectors. This application has been selected since the number of application tasks can be easily parameterizable. Moreover, its low ratio of computation/ communication time guarantees that the results are ***pessimistic***. This means that more complex applications, with longer execution time, would present a relatively lower recovery time. Results

The first experiment, illustrated in Figure 4, shows the impact on the recovery time when the number of reallocated tasks increases. Each scenario executes an application with 8 tasks. The task mapping is such that the defective PE has 1 to 5 tasks before the fault is detected. All affected tasks have the same binary code size. The result shows that the recovery time has a small overhead on the total execution time. The worst case is in the scenario with 5 tasks reallocated where the recovery time was 424.66 µs and the total execution time 6819.15 µs, representing 6.23% from the total execution time. This experiment highlights an important feature of the proposed approach: *the recovery time grows linearly with the number of affected tasks* (r^2=0.992).

The second experiment, illustrated in Figure 5 shows the recovery time when the total size of the application is increased in terms of number of tasks, but the number of affected tasks is constant (3 tasks). The experiment shows that the relative recovery time (dashed line), for all 3 scenarios, is below 4%. *This result shows that the number of tasks of the application does not have a significant impact the recovery time*.

The third experiment, illustrated in Figure 6, shows the impact of the recovery process over the communication traffic. This Figure illustrates the relative bandwidth used in the NoC channels during the fault recovery process. The experiment measures the costs of the three main services related to the recovery (Freeze, Unfreeze, and Reallocation). It shows that the *task reallocation* service dominates the recovery traffic, which is the part of the algorithm that transmits the task's binary code from the repository to a new healthy PE. It means that *the recovery traffic is dominated by the size of the affected tasks*. The worst-case scenario occurs when five tasks are reallocated, with a bandwidth cost of 35.04%. This bandwidth cost represents the

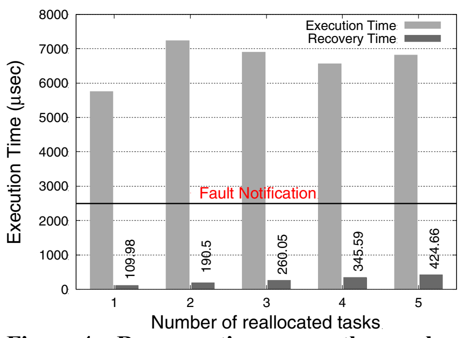

Figure 4 – Recovery time versus the number of tasks reallocated from the failed PE. The fault notification is sent at 2500 µs.

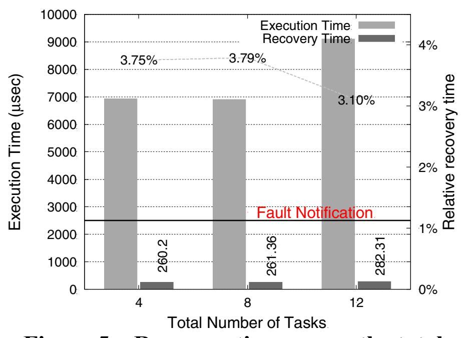

Figure 5 – Recovery time versus the total number of application tasks. The number of reallocated tasks is constant.

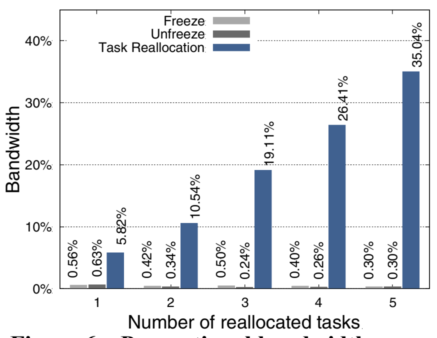

Figure 6 – Proportional bandwidth usage from the main services executed during the recovery process.

peak bandwidth during the recovery process. After the recovery, the bandwidth returns to normal.

A. Limitations

The proposed method has some limitations. For instance, the application *cannot have static initialization of global variables in their declarations*. These variables are located in a memory area which is not modified by the unfreeze step, thus, it would not be able to return these variables to their initial values. This effect happens only for the tasks unaffected by faults, which remain in the same PE. The second limitation is to support *faults at the managers PEs* too.

B. Comparison with Previous Approaches

The method presented by [7] shows hardware and software mechanisms to deal with error detection and recovery management. It has 60% of area overhead, while the proposed method has no hardware cost. The previous method recovers the task only by migrating the context data while the binary codes of all tasks are previously loaded to all PEs. The previous approach does not present the checkpoint overhead to save data context while the proposed approach has no overhead during normal application execution. Due the hardware implementation to support task migration, the recovery time is 0.88% from the total application execution time while the proposed method shows recovery time of about 3.7% using a pure software implementation. Another method shown in [3] presents a self-repair method that re-compiles the code via an off-line repair process, thus, they are not directly comparable to the proposed method. Finally, there are task migration approaches that are used for load balancing [2]. The downtime due the task migration is 1ms (600MHz) for migrating a single task. The proposed method has downtime (i.e. recovery time) of 0.1ms to 0.4ms (100MHz), depending on the number of migrated tasks (from 1 to 5).

V. CONCLUSIONS

This paper presented a fault recovery protocol for MPSoC processors with distributed memory. The results demonstrate that the proposed method has no silicon area overhead since it is software-based. It also has a minor impact on application execution time and bandwidth usage. The impact is *only* during the actual recovery, not during normal operation. The results also show that the recovery time is dominated by the number of tasks (and their respective sizes) affected by the faulty PE. This means that even larger systems with applications with more tasks will have similar recovery time, demonstrating the scalability of the proposal.

ACKNOWLEDGMENTS

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, and FAPERGS - project 2242-2551/14-8. The Author Alexandre Amory is supported by CNPq – project 460205/2014-5.

REFERENCES

[1] Mushtaq, H.; et al. “Efficient software-based fault tolerance approach on multicore platforms”. In: DATE, 2013, p. 921-926.

[2] Bolchini, C.; et al. “Self-adaptive fault tolerance in multi-/many-core systems.” Journal of Electronic Testing, v.29(2), 2013, p. 159-175.

[3] Muller, S.; Sholzel, M.; Vierhaus, H.T. “Hierarchical self-repair in heterogeneous multi-core systems by means of a software-based reconfiguration”. In ARCS Workshops (ARCS), 2012. p. 1-7.

[4] Ter Braak, T. D.; et al. “On-line dependability enhancement of multiprocessor SoCs by resource management”. In: International Symposium on System on Chip (SoC), 2010, p. 103-110.

[5] Amory, A. M.; et al. “Task mapping on NoC-based MPSoCs with faulty tiles: Evaluating the energy consumption and the application execution time”. In: Rapid System Prototyping (RSP), 2011, p. 164-170.

[6] Almeida, G. M.; et al. "Evaluating the impact of task migration in multi-processor systems-on-chip". In: SBCCI, 2010, pp. 73-78.

[7] Derin, O.; et al. “A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project”. Microprocessors and Microsystems, 37(6-7), 2013, 515–529.

[8] DeOrio, A.; et al. “A reliable routing architecture and algorithm for NoCs”. IEEE Transactions on CAD, v.31(5), 2012, pp. 726-739.

[9] Fick, D.; et. al. “A highly resilient routing algorithm for fault-tolerant NoCs”. In: DATE, 2009, pp. 21-26.

[10] Wachter, E.; et al. “Topology-agnostic fault-tolerant NoC routing method”. In: DATE, 2013, pp. 1595-160.

[11] Mandelli, M.; et al. “Energy-Aware dynamic task mapping for NoC-based MPSoCs”. In: ISCAS, 2011, pp. 1676-1679.

[12] Castilhos, G.; et al. “Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes”. In: ISVLSI, 2013, p. 153-158.

[13] Elnozahy, E. N.; et al. “A survey of rollback-recovery protocols in message-passing systems”. In: ACM Computing Surveys Journal, v. 34, 2002, pp. 375-408.

[14] Gizopoulos, D.; et al. “Architectures for online error detection and recovery in multicore processors”. In: DATE, 2011, p. 1-6.

[15] Wang, L-T; Wu, C-W; Wen, X. “VLSI test principles and architectures: design for testability”. Academic Press, 2006.

[16] Psarakis, M.; et al. “Microprocessors software-based self-testing,” IEEE Design and Test of Computers, v. 27(3), 2010, p. 4 - 19.

[17] Kerkhoff, H.G.; et al. “Linking aging measurements of health-monitors and specifications for multi-processor SoCs”. In: DTIS, 2014. p. 1-6.