Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Science Letters:

Low-cost fault tolerance in evolvable multiprocessor systems: a graceful degradation approach

Shervin VAKILI†, Sied Mehdi FAKHRAIE, Siamak MOHAMMADI, Ali AHMADI

(*School of Electrical and Computer Engineering, University of Tehran, Tehran 14395-515, Iran*)

†E-mail: sh.vakili@ece.ut.ac.ir

Received Nov. 19, 2008; Revision accepted Apr. 23, 2009; Crosschecked Apr. 29, 2009

Abstract: The evolvable multiprocessor (EvoMP), as a novel multiprocessor system-on-chip (MPSoC) machine with evolvable task decomposition and scheduling, claims a major feature of low-cost and efficient fault tolerance. Non-centralized control and adaptive distribution of the program among the available processors are two major capabilities of this platform, which remarkably help to achieve an efficient fault tolerance scheme. This letter presents the operational as well as architectural details of this fault tolerance scheme. In this method, when a processor becomes faulty, it will be eliminated of contribution in program execution in remaining run-time. This method also utilizes dynamic rescheduling capability of the system to achieve the maximum possible efficiency after processor reduction. The results confirm the efficiency and remarkable advantages of the proposed approach over common redundancy based techniques in similar systems.

Key words: Fault tolerance, Multiprocessor system-on-chip (MPSoC), Genetic algorithm (GA), Adaptive task scheduling
doi:10.1631/jzus.A0820803 **Document code:** A **CLC number:** TN4

INTRODUCTION

Multiprocessor system-on-chip (MPSoC) is becoming a prevalent trend in designing high performance computation systems. This approach aims to achieve high computational power by concurrent execution of different program segments on different processors while preserving the flexibility, short time-to-market, and simplicity of development of microprocessor-based systems (Martin, 2005). On the other hand, MPSoC faces some major challenges in hardware and software developments. The sequential essence of convenient programming models is the most critical challenge in this area. Decomposition of programs to concurrent tasks and scheduling of these tasks among different processors are two important and difficult steps to overcome this challenge (Wolf, 2004; Martin, 2005).

The evolvable multiprocessor (EvoMP) system is a novel homogeneous MPSoC system suitable for iterative algorithms (Manimaran and Murthy, 1998a). The EvoMP utilizes the network on chip (NoC) for inter-processor communications. The main novelty of this platform lies in the utilization of a hardware genetic core to perform decomposition and scheduling operations dynamically at run-time. The genetic core generates a simple encoded bit-string (chromosome), which is received by all available processors in the system. This bit-string determines the processor, which is in charge of executing each instruction in the program. Thus, EvoMP does not require parallel programming or compile time parallelization; it can directly and efficiently execute single-processor sequential programs in a multiprocessor environment. Low-cost fault tolerance is one of the main features of this platform, which benefits from the absence of centralized controlling units. Feasibility of low-cost fault tolerance is a major advantage of homogeneous architectures. All multiprocessor systems can be divided into two main categories, including static and dynamic scheduling systems. In static scheduling systems, the number of operational processing elements and their assigned tasks must be predetermined. Thus, the fault tolerance can only be achieved by

some dedicated redundant spare processing elements (PEs). The faulty PEs can be replaced by the spares to perform their assigned tasks. For example, Canham and Tyrrell (2003) adopted some spare modules (molecules, which are simple reconfigurable hardware units) in a computational cell in the Embryonic project. Barker *et al*.(2007) used triple module redundancy (TMR) and similar redundancy-based schemes for fault tolerance in bio-inspired POEtic tissue project. In dynamic scheduling systems, similar investigations are mainly concerned with degradable dynamic task scheduling techniques that aim to eliminate the redundant spare components by re-assigning the tasks among available and non-faulty computational resources dynamically (Manimaran and Murthy, 1998b; Beitollahi and Deeconinick, 2006; Obermaisser *et al*., 2008).

EvoMP utilizes a high performance fault tolerance mechanism. Previous works in this field had only proposed techniques to assign the pending tasks to different resources based on predefined patterns. But EvoMP uses the execution time results to improve both decomposition and scheduling schemes at run-time. Thus, it can reach improved results, albeit with a small time penalty for the evolution phase. The EvoMP system and the presented fault tolerance scheme in this letter are fully implemented using RT-level VHDL.

OVERVIEW OF THE EVOMP SYSTEM

As mentioned earlier, EvoMP is a homogeneous NoC-based multiprocessor system with evolvable task decomposition and scheduling. This platform obviously requires enough time for evolution to reach an efficient solution. Therefore, it is suitable for iterative programs like DSP applications that perform unique computations on a stream of data. When the system starts to execute such a program, the genetic core generates random data, which results in random decomposition and scheduling of instructions among the processors. When all processors arrive at the end of an iteration, the genetic core looks at a dedicated counter that counts the clock cycles. In this instance, the counter shows the number of clock cycles taken to execute the entire ended iteration. This value is used as a fitness value for the corresponding chromosome generated by the genetic core. Then, the genetic core generates the next chromosome and the system starts execution of the next iteration with this new decomposition and scheduling scheme. This process is repeated until the genetic core finds an appropriate solution, after which this core goes from Evolution to Termination state (Fig.1b), where the best found result is used as its constant output.

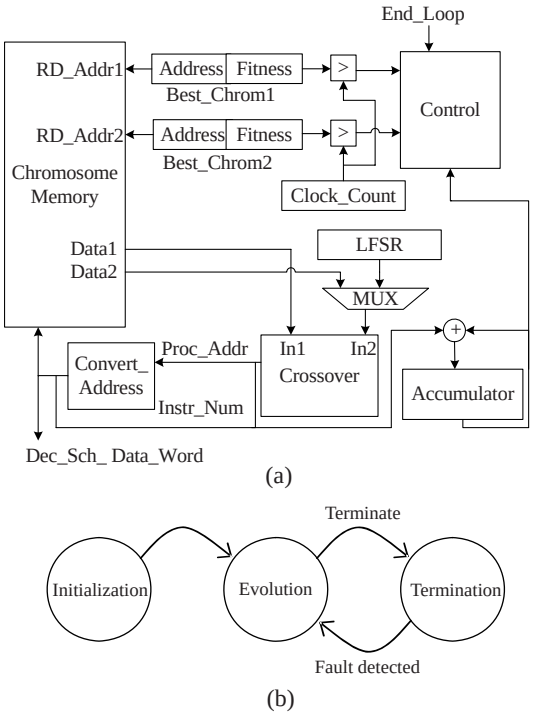

Fig.1 Genetic core. (a) Architecture; (b) State machine

As described above, the EvoMP platform requires no prior information about task decomposition and scheduling of the target program. Therefore, this platform can execute regular sequential programs in a multiprocessor environment efficiently. On the other hand, it needs a primary evolution time to find the best decomposition and scheduling solutions according to the number of available computational resources and the running program.

All processors have one dedicated copy of the program in their internal instruction caches to recognize the processor that is in charge of executing each instruction. Major parameters including the number of instantiated processors are configurable in the EvoMP platform. Vakili *et al*.(2008) presented operational and architectural aspects of the EvoMP platform in more detail.

GENETIC CORE ARCHITECTURE

Genetic algorithm (GA) is the most popular evolutionary algorithm (EA) inspired by genetic evolution of living organisms. In this approach, some candidate solutions, called 'chromosomes', are generated in each generation. These candidates are evaluated by a fitness function unit, and the best ones (elites) are selected. In the subsequent generation, these elites are used by a crossover operator to produce new chromosomes through the recombination process. In this way, the chromosomes gradually move toward better regions of the solutions space. Note that the first generation is chosen randomly (Zomaya *et al.*, 1999).

The presented version of the EvoMP system utilizes a hardware genetic core to perform run-time task decomposition and scheduling. In this system, each candidate solution is encoded as a bit-string, i.e., chromosome. Chromosomes, generated by the genetic core, consist of some scheduling words. Each word determines the processor that will be in charge of executing some successive instructions in the program. A scheduling word contains two fields: the first one identifies a processor, and the second specifies the number of instructions to be executed.

A low-complexity genetic core is designed and exploited in EvoMP. Figs.1a and 1b show the internal architecture of the designed core and its operational state machine, respectively. The system starts to work in the Initialization state, in which the output words are generated randomly using a dedicated linear feedback shift register (LFSR). These words are generated and sent in successive clock cycles at the beginning of each new iteration. The specified number of instructions in each output word is accumulated and the result is compared with the program length. When the accumulator value exceeds the program length, all instructions are scheduled and the chromosome is completed. Note that all output words are also stored in the chromosome memory. Then, this core waits for the End_Loop signal, which is activated only when all contributor processors reach the end of the current iteration. The genetic core uses the Clock_Count counter output as the fitness value of the corresponding chromosome. Two storage components (Best_Chrom1 and Best_Chrom2 in Fig.1a) always store the fitness values and start memory addresses of two best found chromosomes. When the first population is completed, the genetic core goes to the Evolution state in which the best chromosomes of previous populations are used to generate new chromosomes using the Crossover module (that realizes recombination operation). Random numbers are also deployed to generate a portion of each population to ensure that the system will not fall into local optima. If best found chromosomes have not changed for a predetermined number of populations, this core goes to the Termination state (Fig.1b). There are some configurable parameters in this core that strongly affects its performance. These parameters are listed as follows:

(1) Pop_Size: number of chromosomes in each population (generation).

(2) Cross_Rate1: number of chromosomes in each generation generated by crossover between elites.

(3) Cross_Rate2: number of chromosomes generated by crossover between a random data and the best elite.

(4) Rand_Size: number of chromosomes in each generation generated randomly.

EVOMP FAULT TOLERANCE SCHEME

The Convert_Address module (Fig.1a) consists of an Address_Correction module and an Address Mapping Unit (AMU). The Address_Correction module contains a simple logic to convert invalid addresses (e.g., 110, 111 in a 2×3 mesh platform) to valid addresses. The AMU plays an important role in the EvoMP fault tolerance scheme. An internal architecture of this unit is depicted in Fig.2a. As shown in this figure, the AMU consists of a mapping table, a simple controller, and a few simple logic components. After reset, all locations in this memory are filled by their corresponding addresses (i.e., the output is the same as the input). There is also a validation word register in this unit in which each bit indicates the validation of the corresponding address in the mapping table. When a fault is detected in a processor, its address is sent to the AMU. The control module immediately invalidates the corresponding bit in the Validation Word register (Fig.2). An LFSR is used to generate random addresses. The control unit checks the validity of this random address (if it is not valid, it waits until a valid address is generated by the LFSR)

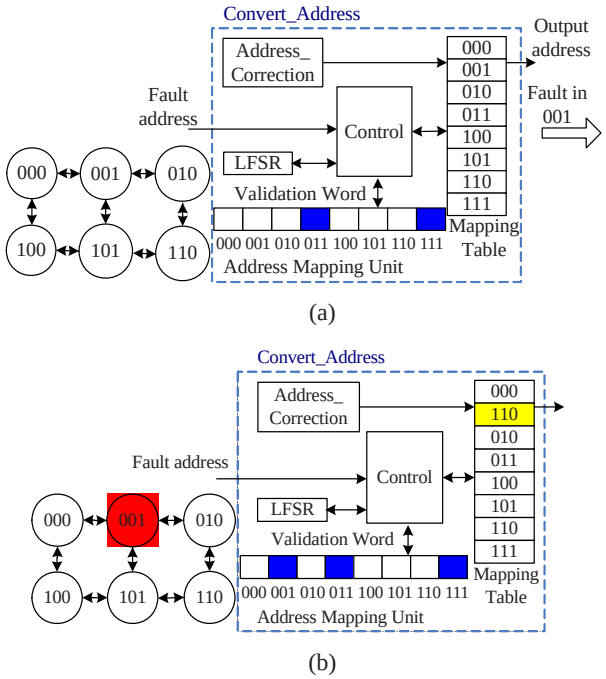

Fig.2 AMU in a 2×3 mesh. (a) All processors are healthy; (b) A fault is detected in the 001 processor

and then replaces all memory locations, which contain the faulty processor address with this random one. In this way, the address of the faulty processor is removed from the entire table and no instruction will be assigned to it in subsequent iterations. It means that the faulty module is eliminated from cooperative computation. The number of computational resources is hence reduced after the occurrence of a fault, and EvoMP faces a new situation. Thus, the genetic core must return to the Evolution state to find the efficient solution for this new situation. The internal architecture of a mapping table unit and the fault tolerance mechanism is shown in Fig.2 (for a 2×3 mesh). While most of the previous approaches can tolerate a limited number of faulty resources, the EvoMP can gracefully adapt itself with any degradation in computational resources. Even with only one non-faulty processor, this platform still stays operational. The genetic core is the only centralized unit in this platform. But as shown in Table 1, this unit is much smaller than the computational section area, and simple redundant hardware techniques (e.g., TMR) can be utilized to make it fault tolerant. Furthermore, note that the genetic core is only important in the evolution phase. Therefore, if it becomes faulty in the Termination state, the system will continue to work without any change. Even if it becomes faulty before the evolution phase (making the evolution impossible), the system will still execute the program, albeit in an inefficient way.

Table 1 Synthesis area results of a 2×2-mesh EvoMP system on an XC2V3000 FPGA

Unit	Area*
NoC switch	741 (2%)
Genetic core	1894 (6%)
Processors	4583 (15%)
Total system	21 877

* The number in the brackets is the percentage of the occupied area out of the total available area

EXPERIMENTAL RESULTS

Some simple DSP programs were used to evaluate the performance of this approach. For this purpose, the system started to work with healthy processors first. After the completion of evolution, the system moved to a termination phase. Then, we injected a fault into a random processor. The system returned to the evolution phase to find an efficient solution for the new situation. Fig.3 shows the best obtained fitness values in such a process for a 2×3 EvoMP system, during the execution of a 16-point discrete cosine transform (DCT) program. The stated process was repeated three times (i.e., three faults were injected into three processors in regular time intervals). As shown in Fig.3, immediately after injecting a fault, the system performance is reduced dramatically.

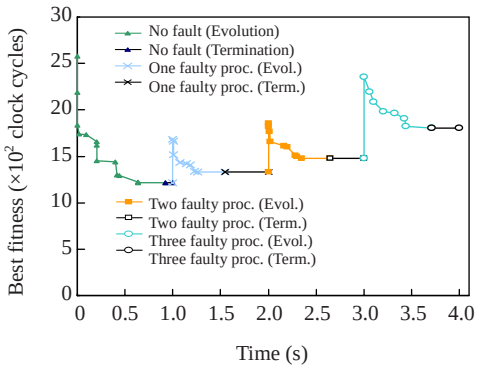

Fig.3 Best obtained fitness (number of clock cycles to execute each iteration) in a 2×3 EvoMP for a 16-point DCT program. Faults are injected into 010, 001, and 101 processors in 1, 2, and 3 s, respectively

The system compensated some portions of this performance reduction through re-evolution. Note that the system continued to work until the last processor failed. Table 2 illustrates the termination phase results and final performance degradations after injecting fault in three processors, during running three different programs. These experimental applications include 8-point and 16-point DCT and multiplication of two 5×5 matrices. These results confirm the applicability and efficiency of the proposed approach in tolerating harmful effects of the probable hardware faults.

Table 2 The final results (clock cycle per iteration) of the evolution phase after randomly injecting three faults into three processors and corresponding percentage of performance degradation in a 2×3 mesh

Program	Number of instructions	Clock cycle per iteration				Percentage of performance degradation			
		n=0	1	2	3	*n*=0	1	2	3
DCT-8	88	285	302	328	406	0	5.9%	15.0%	42%
DCT-16	324	1213	1332	1460	1846	0	9.8%	20.3%	52%
Matrix-5×5	406	1596	1723	1872	2348	0	8.0%	17.2%	47%

n: number of faulty processors

CONCLUSION

The EvoMP is a novel NoC-based MPSoC system with evolvable task decomposition and scheduling. This letter proposes a high performance fault tolerance scheme of EvoMP platforms. This approach simply prevents the faulty processors from contributing in program execution and uses the dedicated dynamic task decomposition and scheduling capability of this platform to gradually adapt itself with the degradation in computational resources. Thus, we can call this system a 'graceful degradable multiprocessor platform'.

The experimental results also confirm the efficiency and remarkable advantages of the proposed approach over common redundancy based techniques in similar systems.

References

Barker, W., Halliday, D.M., Thoma, Y., Sanchez, E., Tempesti, G., Tyrrell, A., 2007. Fault tolerance using dynamic reconfiguration on the POEtic tissue. *IEEE Trans. Evol. Comput.*, **11**(5):666-684. [doi:10.1109/TEVC.2007.896690]

Beitollahi, H., Deeconinick, G., 2006. Fault-tolerant Partitioning Scheduling Algorithms in Real-time Multiprocessor Systems. Proc. Pacific Rim Symp. on Dependable Computing, p.296-304. [doi:10.1109/PRDC.2006.34]

Canham, R., Tyrrell, A., 2003. An Embryonic Array with Improved Efficiency and Fault Tolerance. Proc. NASA/DoD Conf. on Evolvable Hardware, p.265-272. [doi:10.1109/EH.2003.1217678]

Manimaran, G., Murthy, C.S.R., 1998a. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Trans. Parall. Distrib. Syst.*, **9**(3):312-319. [doi:10.1109/71.674322]

Manimaran, G., Murthy, C.S.R., 1998b. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Trans. Parall. Distrib. Syst.*, **9**(11):1137-1152. [doi:10.1109/71.735960]

Martin, G., 2005. Overview of the MPSoC Design Challenge. Proc. Design and Automation Conf., p.274-279.

Obermaisser, R., Kraut, H., Salloum, C., 2008. A Transient-resilient System-on-a-chip Architecture with Support for On-chip and Off-chip TMR. Proc. Int. Dependable Computing Conf., p.123-134. [doi:10.1109/EDCC-7.2008.20]

Vakili, S., Fakhraie, S.M., Mohammadi, S., 2008. Designing an MPSoC Architecture with Run-time and Evolvable Task Decomposition and Scheduling: A Neural Network Case Study. 5th IEEE Int. Conf. on Innovations in Information Technology, p.106-110. [doi:10.1109/INNOVATIONS.2008.4781734]

Wolf, W., 2004. The Future of Multiprocessor Systems-on-chips. Proc. Int. Design Automation Conf., p.681-685.

Zomaya, A.Y., Ward, C., Macey, B., 1999. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Trans. Parall. Distrib. Syst.*, **10**(8):795-812. [doi:10.1109/71.790598]