Prototyping a Fault-Tolerant Multiprocessor SoC with Run-time Fault Recovery

Xinping Zhu
Northeastern University
Boston, MA 02115
xzhu@ece.neu.edu

Wei Qin
Boston University
Boston, MA 02215
wqin@bu.edu

ABSTRACT

Modern integrated circuits (ICs) are becoming increasingly complex. The complexity makes it difficult to design, manufacture and integrate these high-performance ICs. The advent of multiprocessor Systems-on-chips (SoCs) makes it even more challenging for programmers to utilize the full potential of the computation resources on the chips. In the mean time, the complexity of the chip design creates new reliability challenges. As a result, chip designers and users cannot fully exploit the tremendous silicon resources on the chip. This research proposes a prototype which is composed of a fault-tolerant multiprocessor SoC and a coupled single program, multiple data (SPMD) programming framework. We use a SystemC based modeling and simulation environment to design and analyze this prototype. Our analysis shows that this prototype as a reliable computing platform constructed from the potentially unreliable chip resources, thus protecting the previous investment of hardware and software designs. Moreover, the promising application-driven simulation results shed light on the potential of a scalable and reliable multiprocessing computing platform for a wide range of mission-critical applications.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

General Terms

Performance, Design, Experimentation, Verification

Keywords

retargetable simulation, network-on-chip, multiprocessor system, fault-tolerance, system-on-chip, run-time verification

1. INTRODUCTION

Today's integrated circuits (ICs) are becoming increasingly complex. Continued technology scaling enables designers to realize more complex IC architectures on the chips. Thus, it is increasingly difficult to design, manufacture and use these chips. First, designing and verifying these complex ICs has been a daunting task for chip designers. Significant delays, even cancellation, in rolling out large and complex chip designs are not uncommon. Second, chip manufactures are facing huge challenges to manufacture the newer generation of ICs. Increasing the yield of these ICs is a key concern for the semiconductor industry. Third, many reliability issues that exist in the 130nm technology will worsen in the 90nm technology and beyond. For example, soft errors occur when the external radiation or the electrical noise causes state inversion of a flip-flop or memory element. With shrinking feature sizes and decreasing operation voltages, the induced electrical charge becomes more significant. To safeguard against these types of failure, an integrated fault-tolerant IC design approach is needed. Finally, it has become increasing difficult to program these complex ICs due to the inherent parallelism on the chips. Many of the SoC software developers have learned the hard way these SoCs are in fact parallel systems on the chip.

Facing these challenges, this paper proposes a system-level design framework which is composed of a fault-tolerant multiprocessor SoC (MPSoC) prototype and an integrated single program, multiple data (SPMD) programming environment. The proposed MPSoC prototype is a tiled architecture which consists of a number of self-checking processor cores. These tiles are connected by a scalable on-chip network. These on-chip cores communicate to each other via a suite of communication protocols which will guarantee the reliable delivery of on-chip messages. On the software side, we provide an application development environment which is composed of a standard suite of programming tools. In addition, the exact level of parallelism and the fault-tolerant SoC architecture are hidden from the application developers thanks to a software-controlled run-time task allocation mechanism. This run-time virtualization can help designers focus on exploiting the inherent parallelism in the system without detailed knowledge about the precise underlying parallel architectures which may change due to permanent component failures. Consequently, the combination of hardware and software based mechanisms can reliably detect and tolerate both the transient and permanent faults on the chip and ensure the long-term reliability of the chip operation. Using a signal processing application as the benchmark, our simulations shows that this design framework can effectively construct a reliable computing platform from the potentially unreliable chip resources.

The rest of the paper is organized as follows. Section 2 analyzes the fault model of an MPSoC and surveys previous works which this work is built upon. Section 3 examines in detail the hardware and software architecture for such an MPSoC. Section 4 describes the simulation framework and presents the experimental results using digital beamforming as the target application.

2. BACKGROUND

2.1 Fault Model

There exist several dimensions in classifying the possible fault occurrences during the life cycle of an MPSoC. We list the classification as follows:

- *Duration* In terms of duration, the faults can be classified into transient faults and permanent faults [4]. In the case of the MPSoC, both types of fault can occur in the chip life cycle.
- *Location* In general, MPSoC designs consist of two integrated parts, the Processing Elements (PEs) and Network-on-Chip (NoC). Faults can occur in both parts. In the case that a fault occurs in the PEs, the computation results will be erroneous. Dynamic fault detecting and masking actions are needed to make sure the erroneous results will not contaminate the application environment. In the case that a fault occurs in the communication path, such as link failure and scrambled messages, a fault-tolerant communication protocol suite, including error-resilient coding schemes, are needed to ensure the reliable delivery of on-chip messages on top of an unreliable on-chip communication substrate.
- *Time to Failure* Faults can occur throughout the lifetime of an IC. Using the point when the chip is packaged and tested as the watershed event, we distinguish between before-shelf faults and after-shelf faults. Currently, chips with before-shelf faults, i.e., defects which are discovered during testing, are invariably discarded. Only dies with no discovered defects are shipped out as products. With the shrinking feature size, it is becoming increasingly difficult to achieve decent yield with reasonable cost. The low yield problem will become more acute for the 90nm technology and beyond. On the other hand, the potential yield of the manufacturing process can increase tremendously if some defects on the die can be tolerated in the IC's after-shelf life. Static fault masking and isolation techniques, both hardware and software based, can be used to use these previously deemed "bad" chips in commercial products, such as picoChip [6]. For after-shelf faults, dynamic fault detection and recovery means are needed to ensure the correct function of the chip as long as possible. Furthermore, graceful degradation of system performance is necessary for some mission-critical applications.

2.2 Previous Works

The concept of run-time verification is proposed to remedy the increasingly complex verification task facing billion-transistor IC designs. Traditional functional verifications are usually implemented using simulations which only provide a limited coverage of the test space. A more rigorous and formal verification using formal methods is often too complex and sometimes intractable. Run-time verifications use additional verified logic to detect the fault occurrence and recover from the fault during run-time. Several designs such as DIVA and RAZOR have been shown to be effective in ensuring the correct functioning of a pipelined processor with negligible performance degradation and low power consumption [2].

Fault-tolerant computing system designs have been studied extensively [4]. Researchers usually achieve fault-tolerance of multiprocessor based systems using two approaches. In the static redundancy approach, several identical copies of the same program are executed on several processors simultaneously. A majority voting of the replicated results is used to decide the correct computation results. Compared with the majority of the fault-tolerant parallel system designs, our methodology is less conservative and the resulted multiprocessor SoC implementation is more efficient in terms of utilizing the available silicon resources from the new manufacturing technology.

MPSoCs are proposed in the last few years to address the challenge of utilizing billion-transistor scale silicon integrations technology while keeping design and verification tasks manageable. Examples of the MPSoCs include the MIT RAW architecture [7], the picoChip [6]. Within this context, researchers have proposed novel mechanisms to tolerate the faults using fault-tolerance communication protocols [3]. Our work extends these previous analyses by implementing a prototype which can tolerate faults both on the PEs and the communication path.

3. SYSTEM ARCHITECTURE

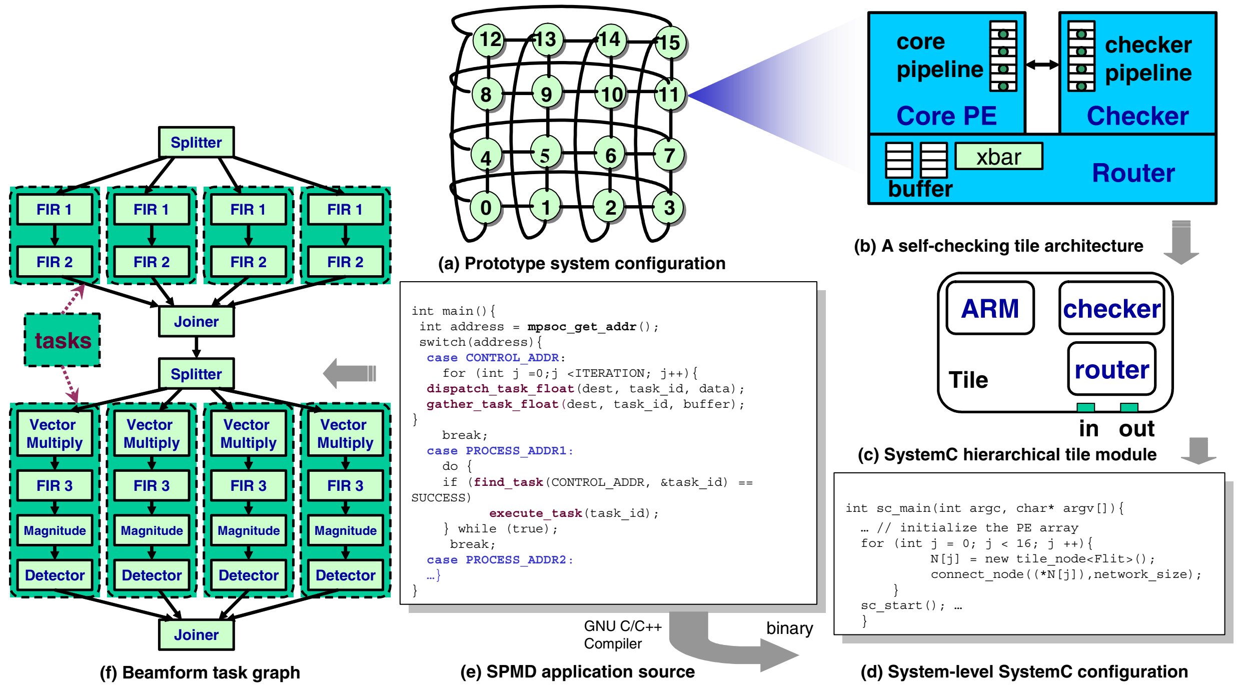
3.1 Hardware Architectures

As shown in Figure 1.(a), the hardware prototype is a tiled architecture which is composed of a configurable number of individual tiles. Each tile is composed of a core PE and an attached checker processor. Figure 1.(b) illustrates the tile architecture. Each PE is an independent processor core which has its own logic and cache. For the purpose of runtime fault-recovery, each PE is attached to a checker processor. The checker processor is a microarchitectural unit which is attached to the commit phase of the processor core's pipeline [2]. With the checker processor, only the correctly computed results will be committed. In the event a transient fault is detected by the checker processor, an exception is raised and necessary steps are taken to correct the erroneous computation results. Researchers have proposed several efficient designs for the checker processor. Some of the designs can provide efficient dynamic run-time fault tolerance with negligible performance cost [2]. Additionally, memory components of the tile can be augmented with efficient error detecting and correcting circuits. Similar to the MIT RAW architecture [7], each tile is capable of communicating to neighboring tiles directly via message passing. A hop-by-hop based credit based flow control policy is used to ensure the buffer availability. This flow control policy can be augmented to guard against message loss through retransmission. The flow control policy provides basic on-chip communication fault-tolerance. Also, efficient coding schemes can be used to protect on-chip links against transient faults [1].

Even though the checker processor can detect and correct transient faults, i.e., soft errors, the occurrence of permanent faults may render the tile either unresponsive or non-reachable. In that case, the prototype needs the capability of detecting the permanent failure of individual tiles and marking them unusable for future computation and communication. To achieve this goal, we propose the adoption of a link state algorithm based approach similar to the basic operations of the Open Shortest Path First (OSPF) protocol.

3.2 Software Architecture

To tolerate the permanent failure of individual tiles in a tiled architecture, there must exist the capability to migrate a task from one tile to another tile. The single program, multiple data (SPMD) is ideally suited for such a context. In the SPMD approach, all the tiles execute copies of the same data-parallel program. In our design, each tile executes a separate portion of the same sequential program according its assigned tasks. If one of the tiles fails, there is no loss of instruction sources because every tile shares the same application binary. The program-level redundancy makes it possible to tolerate tile failures during the execution of the program. Figure 1.(e) shows a SPMD application code snippet.



```
int main(){
 int address = mpsoc_get_addr();
 switch(address){
  case CONTROL_ADDR:
    for (int j =0;j <ITERATION; j++){
  dispatch_task_float(dest, task_id, data);
  gather_task_float(dest, task_id, buffer);
}
    break;
  case PROCESS_ADDR1:
    do {
    if (find_task(CONTROL_ADDR, &task_id) ==
SUCCESS)
            execute_task(task_id);
    } while (true);
     break;
  case PROCESS_ADDR2:
  …}
}
```

```
int sc_main(int argc, char* argv[]){
  … // initialize the PE array
  for (int j = 0; j < 16; j ++){
           N[j] = new tile_node<Flit>();
           connect_node((*N[j]),network_size);
      }
  sc_start(); …
  }
```

Figure 1: The system prototype and the simulation infrastructure

In our approach, the tasks communicate to each other using a set of simplified MPI-1 like message passing library routine calls. Currently, the library is based on high-level languages such as C/C++. Since our parallel model is SPMD based, each PE executes a portion of the same program while in a different context, e.g. processing a different set of data. The unit for these independent portions is called task. Each task is a set of computation iterations operating independently on one PE. A task usually starts by receiving a set of input data from one PE, followed by the desired data processing, and ends by sending back the data to its destination, which could be either the control PE or another task which can continue processing in a pipelined manner. As the first step, we implement a scatter/gather communication based programming model for data-parallel applications. In such a program communication pattern, the data set is initially located in one of the tiles, which is designated as the control tile. The control tile breaks up the data into pieces and distributes the computation tasks to a group of processing tiles. After the computation tasks terminate, the control tile receives the results from the group. The scatter/gather model can be applied to many applications such as space-time adaptive processing, searching, ray tracing, and 3D collision detection. In the current experiment setup, only one tile functions as the control tile and we do not model the fault on the part of the control tile.

To achieve task-level fault-tolerance, we implement a software-controlled dynamic task allocation algorithm. The control tile maintains a pool of tasks. During run-time, before each ready task is allocated to the physical tile, the control tile checks if there is a detected fault in the next-to-be-allocated tile. If a fault has been detected and registered as permanent, the tile will be marked not usable and no further task will be assigned to the specific tile. If the fault is deemed as transient, the specific tile will be conservatively skipped for allocation in the current iteration.

After the task is mapped to the specific tile j and becomes an active task, the control tile checks if the task finishes its computation before a pre-determined deadline, e.g. its task response time threshold $trtt_j$. If the task fails to finish by the deadline, the tile is presumed to be faulty/non-reachable and the unfinished task will be reinserted into the task dispatch queue. The presumed faulty tile will be skipped from the task allocation algorithm. The details of task allocation process are hidden from the application programmer. The programmer is only aware of the façade of a number of virtual processing units. The task allocation routines then automatically bind the virtual processing units to the existing sound tiles during execution, usually in a round-robin fashion.

4. EVALUATIONS

4.1 Simulation Framework

We model the aforementioned hardware and software architecture using a SystemC 2.0 based modeling and simulation framework. The framework has previously been constructed to model a variety of MPSoC prototypes with different configurations [8]. We extend this framework to address the need for a fault-tolerant MPSoC based computing platform. For each tile, we choose a public-domain cycle-accurate ARM simulator as the core PE simulation engine [5]. As shown in Figure 1.(c), the tile module is a hierarchical module whose composition corresponds to the proposed hardware architecture of the physical tile. It consists of an ARM PE module, a checker module and a router module. After implementing the individual tile models in SystemC, the top-level model, as shown in Figure 1.(d), is constructed directly by connecting each tile with customized SystemC channels. The application binary is a multiprocessing binary compiled from C/C++ in the form of a sequential program. Each tile shares the same program while executing a separate portion of the binary depending on the unique PE address, as shown in Figure 1.(e). The compilation is done by standard GNU C/C++ tool chains. No additional compiler augmentation is needed in this case.

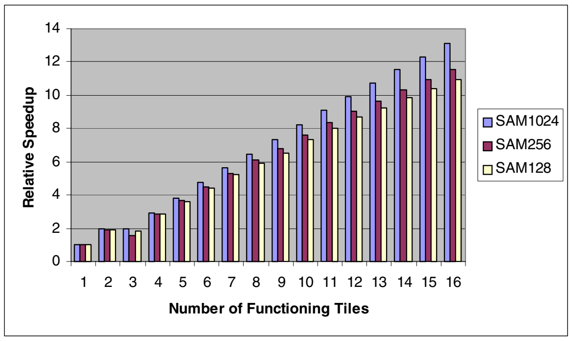


Figure 2: Average speedup for beamforming

4.2 Application

We choose a signal processing application, digital beamforming, as our target application. Digital beamforming is a class of array processing algorithms that optimize an array gain in a direction of interest and/or locate directions of arrival for sources. Novel designs using FPGA and ASIC have been proposed to implement the beamforming algorithms for the next generation of Wi-Fi systems. In addition, wireless data connectivity is often mission-critical and needs "Always On" performance.

The beamforming application is ideally suited to be implemented on an MPSoC. Due to its inherent data parallelism, researchers have proposed parallel implementations of the beamforming algorithms. Our current implementation focuses on the iteration decomposition technique. In this approach, the beamforming job is partitioned across iterations. A number of iterations are clustered together to form a beamforming task. Since each iteration is processing a separate set of array input samples, a single task is operating on one sampled data set, while another task is simultaneously operating on a different data set. Figure 1.(f) shows the task graph of a beamforming application with 4 channels and 4 beams. The shaded regions with dashed borders are the tasks. The number of iterations in each task depends on the number of samples for each channel/beam.

4.3 Simulation Results

We report three sets of application-driven experiments which are conducted on a single MPSoC with different degrees of faulty PEs. Since transient faults can be effectively corrected using the built-in checker processor during run-time, only static permanent faults are modeled in these experiments. These three experiments investigate, first, the impact of PE fault occurrences on system performance; second, the impact of task granularity on system performance.

To compare the results from these three sets of experiments, we maintain the uniformity of the underlying hardware as a 4 by 4 tiled architecture as detailed in Section 3.1. Since the fault can occur at any of the tiles, the location of the faulty tiles will have an impact on the performance degradation induced by the fault. In our simulation, we observe that the impact is negligible for a 4 by 4 mesh configuration and we choose a random PE when a permanent fault needs to be modeled. The reported results show only one of the faulty configurations but the general trend is consistent across all the possible faulty configurations.

The simulations are run on a cluster of Pentium4 2.8 GHz machines with 1M L2 cache, 1GB RAM and 800MHz frontside bus (FSB). The machines run Fedora Core 1 Linux. Figure 2 shows the relative speedup using the cycle counts of the single tile case as the baseline for comparison. Currently, the simulation time ranges from 0.8 to 30 hours. The three sets of experiments are distinguished by the task granularity. The granularity is based on the size of the sample data for each task. The tasks in SAM1024 perform computations for 1024 samples, and the tasks in SAM256 perform computations for 256 samples and the tasks in SAM128 perform computations for 128 samples. The X axis shows the number of the non-faulty tiles available to the beamforming application. In the case of 16 available tiles, no faulty tile exists in the configuration. We observe that the fault-induced performance degradation is smooth and predictable, with the exception of the configurations of very limited number of non-faulty tiles. The reason is that, with three or more functioning titles, one of the tiles is designated as the control tile while the computation tasks are mapped to the rest of the tiles.

The impact of task granularity on system performance is also evident from Figure 2. Experiments with finer tasks, such as SAM128, incur more performance cost, especially in configurations with a large number of sound tiles.

5. CONCLUSIONS AND FUTURE WORK

Given the need for a highly-reliable computing platform for today's increasingly complex and mission-critical applications, we propose a fault-tolerant MPSoC prototype which can reliably detect and isolate faults during run-time. Another contribution of this paper includes a system-level modeling and simulation framework which supports the design analysis and exploration of this platform. Using a data-parallel signal processing application, we show that the prototype can be trusted to execute correctly and scalably in the case of multiple permanent faults on the chip. Lots of exciting work lies ahead. We are working on augmenting this prototype with more run-time support to tolerate permanent and transient control tile faults while meeting real-time constraints. Extending this work to heterogeneous MPSoCs is also on our immediate agenda. Ultimately, we hope this computing platform will be adapted to a wide range of mission-critical application contexts.

6. REFERENCES

[1] D. Bertozzi, L. Benini, and G. De Micheli. Low power error resilient encoding for on-chip data buses. In *Proceedings of 2002 Design Automation and Test in Europe Conference (DATE)*, 2002.

[2] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[3] S. Manolache, P. Eles, and Z. Peng. Fault and energy-aware communication mapping with guaranteed latency for applications implemented on NoC. In *Proceedings of 42nd ACM/IEEE Design Automation Conference (DAC)*, 2005.

[4] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[5] W. Qin. SimIt-ARM. `http://sourceforge.net/projects/simit-arm/`.

[6] W. Robbins. Redundancy and binning of picoChip processors. Fall Processor Forum, 2004, San Jose, CA.

[7] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2), 2002.

[8] X. Zhu, W. Qin, and S. Malik. Modeling operation and microarchitecture concurrency for communication architec tures with application to retargetable simulation. In *Proceedings of International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*, 2004.