Reliability-Aware OS Support for FPGA-Based Systems [Extended Abstract]

Authors: M. Kandemir (PSU) and G. Chen (PSU)

Reconfigurable computing systems have shown the ability to greatly accelerate program execution, providing a high-performance alternative to software-only implementations and a programmable alternative to ASICs. While prior research have addressed architecture design, programming, and compilation issues, there is still not much consensus on what kind of operating system (OS) support should be provided for reconfigurable architectures. Prior OS related work has evaluated different scheduling algorithms on an FPGA based platform, and found that different OS scheduling algorithms can generate different results. Increasing soft-error rates force designers to look at the OS support for FPGAs from a reliability viewpoint as well. In particular, one may want to tailor existing OS services according to the reliability requirements of the execution environment.

Our Approach

The focus of this paper is to design and evaluate a reliability-aware OS scheduler for FPGA based environments. The primary mechanism through which the OS tries to provide reliability is task duplication under QoS guarantees. To achieve this, the proposed approach operates as follows:

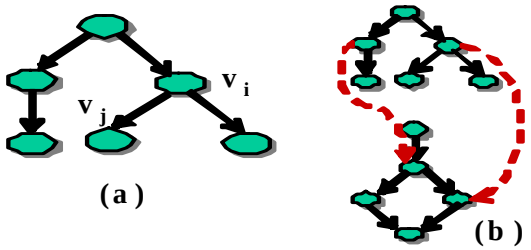

Figure 1. (a) An example STG (note that node v_j depends on node v_i). (b) Two STGs and data dependences between them (shown as dashed arrows).

1) The application programmer indicates which data structures are critical from the reliability viewpoint using annotations (annotation step).
2) The application programmer also indicates the tolerable latency during application execution as a result of the reliability provided (QoS specification step).
3) An automatic application code analyzer analyzes the source code, and identifies tasks (task identification step).
4) Based on how these tasks operate on critical data, they are ranked (i.e., ordered from the most important task to the least important one – task ranking step).
5) The OS scheduler is modified such that whenever there is opportunity, the OS duplicates tasks (that run on the FPGA device) to provide resilience to soft errors (scheduling step).

While our current implementation focuses only on error detection not error correction, it is possible to incorporate error correction into our framework by duplicating a task twice instead of once, and then voting.

We represent each process (task) to be scheduled by a subtask graph (henceforth referred to as STG). Each node of this graph represents a process code portion (subtask) that will be executed in a single quantum of time once it gets scheduled. Note that, depending on the computation being performed by the node, each node may require a different amount of FPGA space (i.e., when it is mapped to the FPGA, it can demand a different size rectangular region than the others). A directed edge (arrow) from a node $v_i \in$ STG to another node $v_j \in$ STG indicates a data or control dependence from v_i to v_j; that is, v_j cannot be executed before v_i (they can be pipelined in some cases; however, we do not consider pipelining in this paper). Figure 1(a) depicts a typical STG for a process (task).

Since one of the objectives of any FPGA-based system is to maximize FPGA utilization, the OS scheduler should be able to schedule nodes from the STGs of different applications. It should be observed that while one may have the option of executing each process in a strict order (i.e., not start executing the next one while the previous one is still running), this may not be a very good idea since dependences between the

nodes of the STG in question would prevent full utilization of the available FPGA space. Therefore, our approach is oriented towards maximizing FPGA space utilization by parallel execution of multiple processes. Also, since our processes are extracted from the same application, there might be data dependences between them (as shown in Figure 1(b)). During the scheduling, whenever the available FPGA space is not fully utilized by concurrently-executing tasks, the reliability-aware scheduler starts duplicating tasks, starting with the most important ones. It also schedules a checker task (per duplicated task), whose sole purpose is the check the outputs of the primary task and the duplicate, and signal an error when the two outputs differ. It is to be noted that just the fact that we have available space on the FPGA for duplications does not necessarily mean that the duplications will not affect the overall execution time. They can still cause performance degradation, due to external runtime conditions and data/control dependences between the tasks that could not be captured by our automatic static analysis (that extracts tasks and dependences between them). Therefore, the proposed scheduler also takes a QoS parameter as input, which indicates the maximum tolerable increase in execution time of the application. Whenever the scheduler predicts that the specified performance degradation limit is about to be reached, it stops duplicating the tasks, and starts acting as a conventional scheduler for the rest of the execution.

Experimental Evaluation

We implemented our approach within a prototype OS and tested it using two applications. We wrote an error injection module, which injects errors with a specified probability. To test the effectiveness of our scheduling strategy, we performed experiments with array-based versions of two large, real-life embedded applications: *encr* and *usonic*. *encr* implements an algorithm for digital signature for security. It has two modules, each with eleven processes (tasks). The first module generates a cryptographically-secure digital signature for each outgoing packet in a networked architecture, User requests and packet send operations have been implemented as processes. The second module checks the digital signature attached to an incoming message. The main data structure used is an array of lists. The application code contains 355 C lines. Our second application, *usonic*, is a feature-based object estimation algorithm. It stores a set of encoded objects, and given an image and a request, it extracts the potential objects of interest and compares them to the objects stored in the database (which is also updated during the process). It is written in C, consists of twelve processes, and contains 830 lines. We executed each application code with both a normal scheduler (Shortest-Job-First) and our reliability-aware scheduler, and measured the percentage of errors caught when our scheduler is used. Since not all the injected errors are harmful, we also collected statistics on the number of errors that would lead to crash of the application (i.e., fatal errors), and how many of these errors the modified scheduler caught. We also collected data on resulting performance degradation. In performing our experiments, we indicated that we can tolerate at most 5% increase in overall application execution time. Also, our current task ranking strategy based on the frequency of accesses to critical data. More specifically, if a task performs more accesses to critical data than another one, we say that the former is more critical (more important) than the latter.

Benchmark	Injected Errors	Fatal Errors	Errors Caught	Fatal Errors Caught	Performance Degradation
encr	62	11 (17.7%)	53 (85.9%)	10 (90.9%)	3.4%
usonic	171	28 (16.4%)	137 (80.1%)	26 (92.8%)	4.1%

Figure 2. Experimental results.

The results from our experimental analysis are given in the table in Figure 2. The second column in this table gives the number of errors injected (with a probability of 10^{-7} per bit per access) and the third column gives the number and percentage of fatal errors. The next two columns give the total number of errors caught by our scheduler and the total number of fatal errors caught. Finally, the last column gives the actual performance degradation experienced. One can see that we catch more than 90% of the fatal errors. Note that this experiment is actually an accelerated test. For example, when we reduce the error injection rate to 10^{-9}, our scheduler was able to catch all the fatal errors. Similarly, when we increase the tolerable performance degradation to 10%, we were able to catch all the fatal errors, even with the error injection rate of 10^{-7}.

Our ongoing work includes experimenting with a diverse set of benchmarks, and implementing task duplication within other types of OS schedulers such as First-Come-First-Serve.