FPGA Hardware Implementation of Statically-derived Application-aware Error Detectors[*]

Peter Klemperer[†], Shelley Chen[§], Karthik Pattabiraman[†], Zbigniew Kalbarczyk[†], Ravishankar K. Iyer[†]

[†]*Center for Reliable and High Performance Computing*
University of Illinois (Urbana-Champaign)
{klempere, pattabir, kalbar, rkiyer}@uiuc.edu

[§]*SAIC*
Champaign, IL
shelley.chen@saic.com

Abstract

Previous software-only error detection techniques have provided high-coverage, low-latency detection but suffer significant performance overheads with a large percentage of benign detections. This paper presents a FPGA hardware implementation of application-aware data error detectors. The detectors are automatically derived at compile time and executed in hardware at runtime, minimizing the performance overhead. We implement the static detectors using the Reliability and Security Engine, which provides a standard interface for developing reliability and security hardware modules. An initial, proof-of-concept model shows that there is only a 2% performance penalty when the detectors are implemented in hardware.

1. Introduction

This paper presents a hardware implementation of an error detection technique to protect applications against data errors. Traditionally, this has been done through duplication-based techniques. Software-only duplication techniques, such as [1], perform checking after every instruction in order to achieve high coverage. Unfortunately, considerable performance needs to be sacrificed for this high coverage. Other techniques reduce the granularity of the checking in order to regain performance [10][7]. However, reduced checking has insufficient coverage, as it cannot detect errors such as hangs and crashes that occur between checks. The IBM G5 moves the comparison logic into hardware in order to reduce the performance overhead of duplication [11], but this results in high hardware design complexity and area overheads. In addition, full duplication, whether implemented in hardware or software, detects many benign errors [5], or errors that do not impact the final outcome of the application. Ideally, we want a solution where high coverage is maintained, with low performance and hardware area overheads.

[9] details a technique which generates application-aware error detectors to protect a program from data errors. Protecting structures like the register file and memory with well-known techniques such as ECC is insufficient as data errors can result from incorrect computation. The detectors are derived statically through compiler enhancements and execute the checks at runtime. Initially, the checkers were implemented entirely in software, resulting in an average performance overhead of 33%, but maximum overheads of up to 80%. The path tracking was also implemented in software, resulting in intolerable overheads (400%). They proposed moving the implementation into hardware, which can significantly improve the efficiency of the technique.

This study utilizes the same technique presented in [9] to statically generate the detectors, but implements the checker logic in the FPGA-based hardware. We see substantial performance benefits for a small application with very simple checker logic. With larger programs, which need more complex checking expressions, even greater benefits are expected.

The rest of this paper is organized as follows: Section 2 gives some background information. Section 3 dives into the implementation details of the detector module. Section 4 evaluates the hardware module and Section 5 describes limitations of the module and some future work.

[*]This work was supported in part by the U.S. Department of Commerce under Grant SBAHQ-05-I-0062, NSF grant CRI CNS 05-51665, Gigascale Research Center (GSRC/Marco), Motorola Corporation, and Intel Corporation.

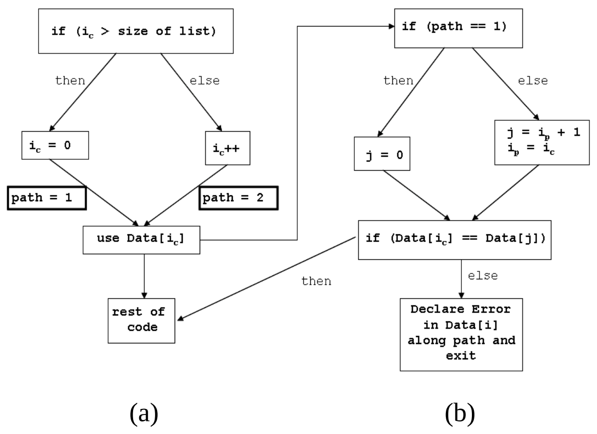

Figure 1. Backward slice of critical variable, $Data[i]$, from inner loop of bubblesort. (a) original code. (b) checking expressions.

2. Background Information

2.1. Path Tracking and Checkers

This section gives a high level overview of the static derivation of the error detectors, which are invoked at runtime. Further details of the technique can be found in [9].

First, we identify the critical variables for the target program. These are variables that are highly sensitive to random data errors. In [8], we performed a study to identify sensitive variables in a program. In that study, we found that variables having high fanout (successors in the dynamic dependence graph) are highly sensitive to data errors. This is because an error in a high-fanout variable can propagate to many other variables and result in program failure (crash or incorrect output). Other metrics that were considered in the study include variables with high lifetimes (distance between their definition and use) as well as variables with high execution count (number of times the instruction computing the variable is executed). However, these metrics resulted in lower error-detection coverage compared to variable fanouts. Therefore, protecting fanout variables will result in maximum error detection coverage.

The fanouts for each variable can be determined through application profiling and analysis of the application's data flow by the compiler. The compiler first chooses the variables with the highest fanouts in the program. Next, the compiler computes the backward slice for the critical variables. A backward slice of a particular variable at a program location includes any instruction that can affect the value of the variable at that location [12]. Heuristics are implemented to keep the memory storage required for the dependency trees low, which allows for scaling to larger applications. At this point, the compiler computes the checking code (later implemented in hardware) and inserts it into the program. Recall that the checks are path-specific. Hence, path-tracking instrumentation is added by the compiler to the application to ensure that the correct checks execute at runtime.

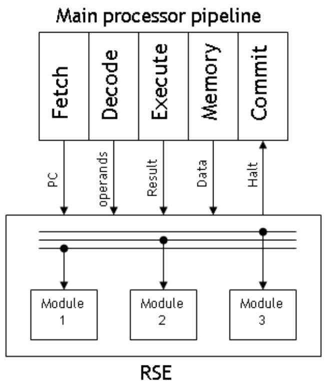

Figure 2. Block diagram of host with RSE.

Figure 1 illustrates the statically-derived detectors. The original code is shown in Figure 1(a). It shows the inner loop of a simple bubblesort application. The derived checking expressions are shown in Figure 1(b). Assuming the compiler has identified the $Data[i_c]$ as a critical variable, it creates checking expressions to recompute $Data[i_c]$ in $Data[j]$ through i_p, a stored value of i from the previous iteration. Afterwards, the values in $Data[i_c]$ and $Data[j]$ are compared. If they are different, an error is thrown. Otherwise, normal program execution continues.

In [9], the path tracking and the checks are executed in software, resulting in a significant performance reduction versus the unprotected application. In our design, the path tracking and the checks are moved into hardware so that the overhead at runtime is minimal. We implement both the path tracking and checks as a module in the Reliability and Security Engine, which is detailed in the following section.

2.2. Reliability and Security Engine

The RSE (Reliability and Security Engine) [3][4] is a framework that provides a standard interface between the host processor and the modules that implement reliability and security services for the executing application. Figure 2 illustrates a block diagram of the RSE connected to host processor. The modules are running alongside the host, monitoring the behavior of the executing application. Probes are inserted into the pipeline of the host processor and continuously transfer all the host state information to the RSE modules. Each module needs only a subset of the overall state information. Thus, some routing functionality is integrated into the RSE to get the proper signals to the proper modules.

The RSE provides an ideal interface for the development of new reliability and security modules. Many probes have already been inserted into the host processor pipeline. Thus, no further intrusions need to be made as these signals are already being forwarded from the host to the RSE. In order to design a new module, one would just need to create the module and route the necessary signals to it.

In addition to the RSE being aware of the application running on the host, the application itself must be aware of the modules that are in the RSE. We augment the ISA with CHECK instructions that enable and disable the different modules. The software developer must indicate to the compiler what parts of the application he wants to protect and the compiler inserts the necessary CHECK instructions to enable and disable the corresponding modules.

2.3. Fault Model

Our fault-model covers any hardware fault that results in incorrect data values (data errors) in the program. Such faults include but are not limited to:

- **Faults in Fetch and Decode Units**: Either the wrong instruction is fetched, and it writes to an active register/memory location in the program (OR) a correct instruction is decoded incorrectly, either resulting in an incorrect (but valid) opcode, or resulting in an incorrect register/memory operand being written to or read from the instruction.
- **Faults in Execution and Load/Store Units**: An ALU instruction is executed incorrectly inside a functional unit, (OR) the wrong memory address is computed for a load/store instruction.
- **Faults in Cache/Memory/Register File**: A value in the cache, memory, or register file experiences a soft error that causes it to be incorrectly interpreted in the program (assuming that ECC is not used).

All the above three categories of faults result in corruption of architectural state. Faults that do not affect architectural state usually raise an exception in the same instruction in which they occur and would be immediately detected by the processor itself. Hence, we do not consider these faults in our fault-model.

3. Implementation Details

3.1. Software Application Instrumentation

The static-detector has been implemented as a module in the RSE on a superscalar DLX microprocessor [2]. For now we have hand-instrumented a sample application, bubblesort, as a proof-of-concept design. In the future, the instrumentation process can be automated using tools already developed for software instrumentation, but not yet adapted to the DLX ISA and the hardware modules.

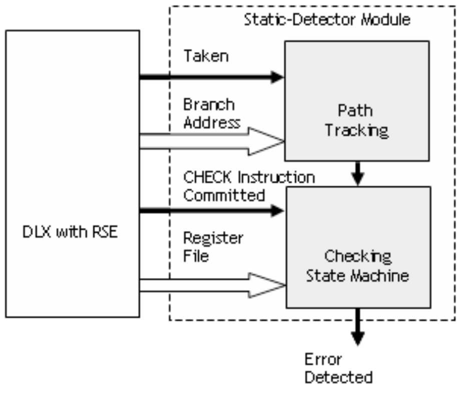

Figure 3. Static-Detector block diagram.

For the bubblesort application, we show a code segment for the inner loop iteration in Figure 1. We deemed the first data value of the sorting algorithm ($Data[i_c]$) as the critical variable. Figure 1(a) shows the selected instructions which affect the calculation of the critical variable, $Data[i_c]$, included in the backward slice of the control flow for a single iteration.

Two parameters must be provided to the static-detector: *critical branch locations* and *state machine definitions*. The PC of the critical branch is required for path tracking. State machine definitions describe the actual checks to be performed. These parameters are all derived from the specific details of the application and its compiled implementation.

3.2. Static-Detector Module Hardware Implementation

The hardware implementation of the static-detector module uses the interfacing capabilities of the RSE to access the internals of the DLX host processor which are both synthesized together onto the FPGA fabric. Figure 3 illustrates the inputs and the outputs to the detector module. There are two main components to the static-detector module: the *path tracker* and the *checkers*.

The path tracking section of the static-detector module utilizes signals from the branch resolution unit of the DLX. The path trackers keep track of which branch path is taken during program execution. The inputs required are the PC of the critical branch and the result of the branch (taken or not taken). For this implementation, the PC is input at module synthesis time and stored in a ROM. Several improvements are discussed in the future work section.

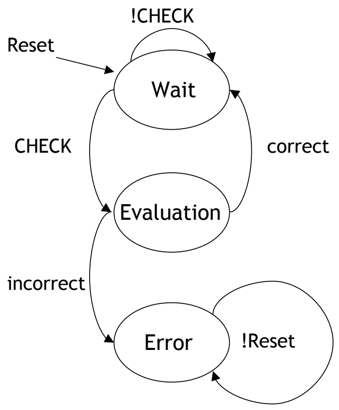

Figure 4. State Machine for Checkers.

The checkers are the code segments that verify the recomputed results for errors, as shown in Figure 1(b). They use the output of the path tracker to determine what checking code needs to be executed. In this implementation, one state machine was hand-derived to perform the checking operations and synthesized into the module. The state machine takes the value recomputed in the checking state-machine and compares it to the result originally computed by the host. If the two values do not match up, then an error has been detected.

The state-machine template consists of a *Wait* state, an *Error* state, and an *Evaluation* state for each of the possible checks. This is illustrated in Figure 4. The *Wait* state waits for a CHECK instruction to be committed by the DLX pipeline. When a CHECK is detected, the branch tracker is consulted and the state-machine transitions to the *Evaluation* state of the indicated branch. An *Evaluation* state performs the appropriate checking expression. If the evaluation of the expression indicates an error, the state-machine transitions to the *Error* state, otherwise, if the expression evaluates correctly, the state-machine transitions back to the *Wait* state. If the *Error* state is reached, the state-machine flags such to the processor and will not transition out of the *Error* state until a processor reset is performed.

Currently, the checks are hand-derived from the software checks by extracting the expressions that form the basis of each check. The checking expressions are rewritten as VHDL expressions and inserted into the state-machine template.

4. Evaluation

The hardware implementation of the static-detector module utilizes a Nallatech BenONE PCI card with a Xilinx XCV2VP70 Virtex-II Pro FPGA chip [6]. We use the Xilinx XST tool to synthesize the designs and the Xilinx ISE tool flow for bitfile generation. In addition, Xilinx Chipscope Pro software logic analyzer is used for testing of the hardware implementation.

Software checks have traditionally run slowly because they add a significant amount of overhead to the application code for implementing the checks. By implementing the checks in hardware, we are able to accelerate the program considerably. This comes from both moving the logic into the hardware, and being able to run the checks in parallel with the execution of the application on the DLX pipeline. As shown in Table 1, with a bubblesort application, we are able see overall speedups of 352% over a pure software instrumentation of the technique.

Faster acceleration is possible by allowing the RSE direct access to main memory. In this implementation, any checks that require access to main memory are loaded directly from memory by the RSE interface. Our tests show that this reduces the performance overhead of the checkers to only 2% over an unprotected design, which is a 345% speedup over our software-checking-only implementation, as shown in Table 1. The remaining 2% overhead represents the effect of the CHECK instructions that are still present in the instrumented code. Bubblesort has a very tight main loop, thus the CHECK instructions account for a significant percentage of the instructions in the inner loop. In addition, the overhead of path tracking is minimal with the hardware implementation. This is because no specialized instructions are necessary for tracking.

If DMA from the RSE is not available, the main processor pipeline can load the critical information from memory and store it into registers accessible by the static-detection module. In this case, overheads of 90.9% were encountered in our testing, but this still represents a speedup of 138% over a pure software implementation. This cost represents the overhead of the processor loading main memory within the program instead of the RSE loading from main memory in parallel with the program. For the bubblesort application, only two memory locations are loaded, but with more complicated programs this overhead can be very high.

The DLX Processor runs at 4 MHz for testing purposes. Table 2 illustrates the area requirements of the design. The

Table 1. Bubblesort Performance Evaluation

Evaluation	Cycles	Performance Overhead
No Instrumentation	30,067	-
Static-Detector Module w/ DMA	30,688	2.1%
HW Static-Detector Module	57,411	90.9%
SW Static-Detector Module	136,607	354.3%

static-detector module requires only an extra 271 slices, 2% of the area of the synthesized superscalar DLX and less than 1% of available chip area for the XC2VP70 FPGA. There is a negligible difference in maximum clock rate when the static-detectors are included in the design, indicating that the module does not create any critical paths that would slow the overall design.

The Static-Detector Module (SDM) operation is verified using fault injection. We inject both control and data faults into the operation of the bubblesort program and verify that the checker does detect the errors. Some example faults follow.

- *Control flow fault*: Control flow faults, such as a miscalculated branch target, can result in the omission of instructions from a program run. For this experiment, the omitted instructions are replaced with NOPs (inside the instruction-cache) during a run of bubblesort. The specific instructions omitted from bubblesort determined which two values in the list are sorted in a given iteration. Having skipped these instructions causes the list to be improperly sorted. The checking functions inside the Static-Detector Module detect that the sorting had not completed properly, and an error is flagged.

- *Data fault*: The data fault injection was carried out by changing the memory values of the list of value to be sorted, in between the time when the sorting function finishes writing back to the list and the hardware checks are carried out. The checker was able to detect a difference between the data that had been written back to this memory line and that memory's state at the time of the check, thereby indicating an error.

The Static-Detector module is able to detect both the control and data errors injected into the operation of bubblesort with minimal impact on performance and area overhead. In this paper we do not perform rigorous fault-injection experiments to assess the coverage of the proposed technique, but [9] presents such an evaluation of the technique implemented in software. We believe that the coverage results for the technique implemented in hardware are likely to be similar to those presented in [9]. Future work will involve injecting a wider range of hardware errors to evaluate the technique

Table 2. Static-Detector Synthesis Stats

	Slice Utilization	BRAM Utilization	Max Pin Delay
DLX + RSE	12262/33088 (37%)	45/328 (37%)	13.134 ns (76 MHz)
DLX + RSE + SDM	12533/33088 (37%)	45/328 (37%)	13.009 ns (77 MHz)
DLX + RSE + SDM + ILHD + PTaint	12500/33088 (37%)	45/328 (37%)	16.099 ns (62 Mhz)

5. Limitations and Future Work

The proof-of-concept bubblesort design demonstrates the feasibility of implementing the static-detector module in FPGA hardware, but the design still has many obstacles to overcome before it can be integrated into large-scale applications using automated tools. These issues are discussed in the next few paragraphs.

The path tracking implementation currently in use is somewhat limited. It can currently only track one branch at a time and requires the PC of the critical branch to be synthesized into the design. Future designs will require tracking multiple paths. Additionally, the PC of the branches should not have to be known until the program load-time. One solution is to add special CHECK instructions which would contain the PC of the critical branches encoded within them and load them with other program initialization data. Furthermore, we imagine creating state machines which contain all valid paths through the critical branches and would allow more complicated tracking schemes to be implemented [3]. These state machines could also be loaded using special loading CHECK instructions.

A similar, but more daunting, problem is also presented in implementing expanded state-machines for the static-detector module's checker routines. One approach is to encode the state machines into a standardized format and load them into register arrays within the static-detector module. This may have high overhead in on-board memory and loading time if the modules are loaded using encoded CHECK instructions or even direct memory access. Another approach we are investigating is automatically generating custom VHDL descriptions for each check and synthesizing unique checking state-machines for each software application. This approach will likely create high-performance solutions with minimum area overhead over a hand-designed solution.

6. Conclusion

The Static-Detector Module in the Reliability and Security Engine offers promise of high-performance, application-aware checking with low logic overhead. This FPGA implementation demonstrates that statically-derived detectors placed in hardware offer significant acceleration over software-only techniques and that marginal overhead over un-instrumented software will be possible in the near future.

References

[1] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, page 71, Washington, DC, USA, 2000. IEEE Computer Society.

[2] J. Horch. DLX processor. Online. http://www.rs.tu-darmstadt.de/downloads/docu/dlxdocu/SuperscalarDLX.html.

[3] R. K. Iyer, Z. Kalbarczyk, K. Pattabiraman, W. Healey, W.-M. W. Hwu, P. Klemperer, and R. Farivar. Toward application-aware security and reliability. *IEEE Security and Privacy*, 5(1):57–62, 2007.

[4] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An architectural framework for providing reliability and security support. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 585, Washington, DC, USA, 2004. IEEE Computer Society.

[5] N. Nakka, K. Pattabiraman, and R. K. Iyer. Processor-level selective replication. In *Proc. of Intl. Conference on Dependable Systems and Networks (DSN)*, 2007.

[6] Nallatech. Bendata-II product brief. Online. http://www.nallatech.com/mediaLibrary/images/english/3576.pdf.

[7] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, pages 63 – 75, March 2002.

[8] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of 11th International Symposium on Pacific Rim Dependable Computing (PRDC)*, pages 75–82. IEEE Computer Society, December 2005.

[9] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *Proc. of Intl. Online Testing Symposium (IOLTS)*, 2007.

[10] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[11] T. J. Siegel, E. Pfeffer, and J. A. Magee. The IBM eServer z990 microprocessor. *IBM J. Res. Dev.*, 48(3-4):295–309, 2004.

[12] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.