An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors

Galen Lyle, Shelley Chen†, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar Iyer

Center for Reliable and High Performance Computing
University of Illinois (Urbana-Champaign)
{glyle, pattabir, kalbarcz, rkiyer}@illinois.edu

†*SAIC*
Champaign, IL
shelley.chen@saic.com

Abstract

Critical Variable Recomputation (CVR) based error detection provides high coverage for data critical to an application while reducing the performance overhead associated with detecting benign errors. However, when implemented exclusively in software, the performance penalty associated with CVR based detection is unsuitably high. This paper addresses this limitation by providing a hybrid hardware/software tool chain which allows for the design of efficient error detectors while minimizing additional hardware. Detection mechanisms are automatically derived during compilation and mapped onto hardware where they are executed in parallel with the original task at runtime. When tested using an FPGA platform, results show that our approach incurs an area overhead of 53% while increasing execution time by 27% on average.

1 Introduction

The continual shrinkage of transistor process technology has increased the need for more reliable systems. Resulting smaller capacitances have increased the likelihood of transient or soft errors which impact application availability [1][9]. Though many existing software solutions, such as duplication [2][11][14][15], provide the necessary reliability, they suffer from being overly conservative and inefficient as many of the detected errors are benign [8].

These conservative detection strategies often incur high overheads either in execution time or in additional hardware complexity [15]. These impediments must be overcome in order to assure the likely adoption of a technique. This is especially true for applications that are time critical, such as medical applications, or which are limited in the amount of available hardware resources, such as embedded systems. Thus, it is essential to minimize the performance and area overheads of additional detection mechanisms.

Toward this, our recent study presented an intelligent recomputation technique called Critical Variable Recomputation (CVR) in [13]. Under this technique, the compiler profiles an application to determine which data is most critical. Recomputation is then performed only on data deemed critical to each application. However, when implemented in software, the technique incurs a considerable performance overhead requiring up to 141% and 555% more time to complete when run on the Pentium 4 and Leon3 platforms, respectively.

To address this undesirable overhead, we move the bulk of the detection mechanism into hardware which is significantly faster and far more efficient than software. The software implementation is limited by the availability of hardware resources. Thus, the checking code must compete against the application that it protects for processing cycles which may stall program execution. With dedicated hardware, however, we can execute checking operations in parallel with the original application, significantly reducing execution time. Moreover, the hardware implementation requires less area overhead and complexity than full duplication.

Unfortunately, many microprocessors are so complex that it is difficult to sufficiently validate all aspects of the design. As a result, extraneous features, such as fault detection hardware, are often poorly tested. For example, Intel recently released a specifications update for their Core processors and most of the errata consist of malfunctioning exceptions and other fault detection hardware [4]. Many of the workarounds consist of disabling the detection hardware, enabling the core hardware to function properly without the checking mechanism. On the other hand, the hardware checks we propose are very simple and involve the execution of common operations such as additions and subtractions. This greatly simplifies the validation process of the design.

In this paper, we present a hybrid hardware/software tool chain which derives and implements low-cost (in terms of performance and area overhead) application-specific error detectors. Analysis of the source code for target applications is performed statically at compile time in order to retain all necessary characteristics of the program. Enforcement of the checks is maintained in hardware to minimize the detection latency and error propagation.

The contributions of this paper include:

1. The design, implementation, and evaluation of a low latency hardware checking mechanism.

2. An automated tool chain for generation of the hardware checks. The tool chain is like a compilation step which facilitates easy adoption.

2 Related Work

A variety of techniques have been proposed for detecting errors in programs. Static analysis techniques typically validate applications based on knowledge of common programming bugs (e.g. NULL pointer dereferences). Dynamic analysis techniques derive code-specific invariants based on dynamic characteristics of the application. However, subtle errors such as timing errors (e.g. race conditions) may still persist in a program [6]. Runtime error detection techniques, such as software duplication, which duplicates the program at the source level [2], instruction level [11], or the intermediate code level [14], are geared toward addressing both these and other software and hardware errors. However, these techniques usually suffer from very high overhead. In order to reduce the overhead, there is a need for runtime error detection techniques that are (1) customized to the application's characteristics and (2) able to detect errors that matter to the application (i.e. errors which, if undetected, lead to failure).

Conversion of C code into synthesizable VHDL has been extensively studied in literature [17][3]. However, such techniques are generic and are thus inefficient in a specialized context. The technique proposed in this paper allows for more efficient hardware synthesis compared to these generic techniques because (1) the checking expressions derived by the technique are devoid of loops/control-transfers, (2) the checking expressions have a common high-level structure, and (3) the detectors do not interfere with normal program behavior.

3 Critical Variable Recomputation Background

The Critical Variable Recomputation (CVR) technique was first presented in [13] where it was fully implemented in software [1]. The goal of CVR is to achieve near-duplication levels of error-detection coverage with only a fraction of the performance overhead. This is feasible because the technique focuses on the recomputation of *critical variables* which are highly sensitive to random errors in the program. By protecting only a few of these critical variables, it is possible to achieve high coverage for errors that are likely to lead to program failure.

The basic steps followed in the design and implementation of the CVR technique are as follows.

1. Determine the critical variables in the program. These variables are chosen based on their *fanout* (defined as the number of forward dependencies); those with a higher fanout are deemed to be more critical. The most critical variables are then chosen for protection. More details concerning this selection is described in [12].

2. Derive the backwards slice, or the dependency trees, of the critical variables. The backwards slice consists of those instructions that can legally modify the critical variables according to the C language semantics [16].

3. Create path tracking and checking expressions for critical variables by optimizing the backward slice for each path which contributes to the computation of a critical variable.

4. Convert the path tracking and checking expressions into state machines which are programmed into reconfigurable hardware. Instrument the original program with special CHK instructions which trigger state transitions when control paths must be tracked or when recomputation must occur.

5. Compile the instrumented application onto a generic processor. The path tracking and checking state machines are programmed into reconfigurable hardware.

4 Hardware Implementation

This section discusses the design and implementation of the CVR hardware.

4.1 Reliability and Security Engine

Our work is built on top of the Reliability and Security Engine (RSE) [10], a framework that provides a general interface for the modular development and implementation of reliability and security techniques in hardware. This approach enables the development of isolated hardware modules for reliability and security enhancements without sub-

[1] While limited hardware support was assumed in the paper, the technique itself was demonstrated exclusively in software

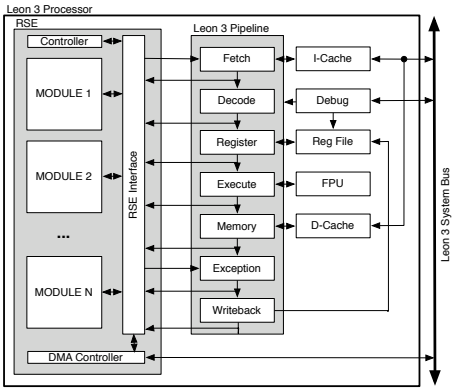

Figure 1. The Leon 3 processor with the Reliability and Security Engine

stantially modifying the host processor. Moreover, the modular environment allows users to customize their system by enabling or disabling techniques as they wish.

We have integrated the RSE with the Leon 3 open-source processor pipeline from Gaisler Research [5], depicted in Figure 1. Although shown with the Leon 3, the RSE framework is general enough to be integrated with any general purpose processor. Additionally, the RSE is non-intrusive to the main processor pipeline as it only needs to monitor execution behavior. This is achieved by inserting probes into the pipeline of the host processor which continuously transfer select state information to the RSE modules.

In order to allow for communication between the application running on the host processor and the RSE modules, we override the SPARC v8 instruction set architecture coprocessor operation instruction (CPOP1) and convert it into a CHK instruction. From the point of view of the main processor, a CHK instruction is viewed as analogous to a no-operation instruction. Each CHK instruction has a unique type identification which specifies the RSE module and operation for which it is intended.

4.2 The Static Detector Module

The Static Detector Module (SDM) is the hardware implementation of the CVR technique described in Section 3. As shown in Figure 2, the SDM consists of two submodules: (1) the **Path Tracking** submodule, and (2) the **Checking** submodule. Typically, data relevant to both submodules is provided in the CHK instruction word. If necessary, an argument buffer, called the ARGQ, buffers data supplied by an SDM-protected application in order to support recomputation involving several operands. Once all values necessary for recomputation are supplied, the ARGQ is accessed in a manner similar to a queue which allows the SDM to perform ordered operations without requiring further information from the application.

Path Tracking. The Path Tracking submodule tracks the location within the application where execution is occurring. This dynamic state information is needed to indicate *which* operations to recompute. Each path corresponds to a different check in the Checking submodule.

The Path Tracking submodule consists of hardware state machines and a stack structure (StateStack). Each state machine corresponds to a particular check and is constantly updated during program execution. The StateStack is implemented as group of individual stacks, one for each state machine. Partitioning the stack in this manner offers major benefits over a unified stack. Specifically, the complex routing logic and wiring associated with maintaining a unified stack yields unsynthesizable designs. Additionally, the overhead associated with stack accesses is minimized as each state machine may access its own stack in parallel with other state machine accesses. The Path Tracking submodule recognizes three different types of CHK instructions:

- *emitEdge(src,dest):* This is invoked whenever a branch is encountered during program execution. The arguments for this instruction (the source and destination values) are inserted into the ARGQ by the application. Since branches are always merging and splitting, the technique requires both the source and destination. The path tracking state machines are then updated accordingly based on these arguments.

- *enterFunc:* This is invoked when program execution enters a function. At this point, the current state of all state machines are pushed onto the StateStack and are cleared.

- *leaveFunc:* This is invoked whenever program execution returns from a function. The state machines are restored to their previous states, which are popped off of the StateStack.

Checking. The Checking submodule performs recomputation operations in parallel with the program execution.

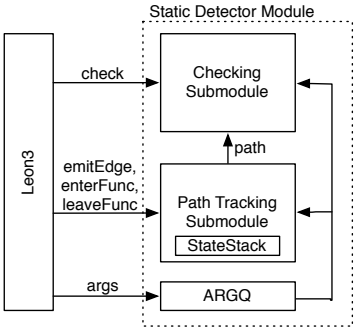

Figure 2. Static Detector Module block diagram

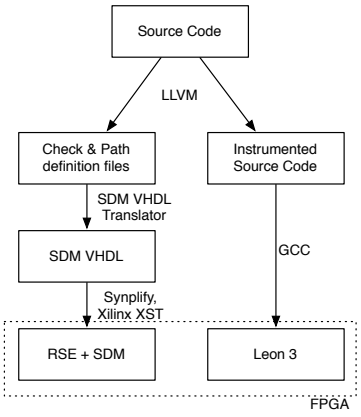

Figure 3. Tool chain for the generation of hardware checks

Checks are invoked with CHK instructions which are embedded into the program source code. This module is responsible for figuring out *when* recomputation takes place. The Checking submodule recognizes only one type of CHK instruction:

- *check(num):* This is invoked when the check is to be executed. The *num* argument indicates the ID of the check that is to be invoked. The Checking submodule also uses information output from the Path Tracking submodule to execute the correct version of each check. This information is needed for cases such as a loop since the check to be performed may depend on whether the program execution is looping or exiting the loop.

After recomputation, the Checking submodule performs a comparison between the normal application computation and the hardware recomputation. If the two computed values differ, the module throws an error. As with the Path Tracking submodule, the Checking submodule gets arguments that it needs either for recomputation or comparison from the application through the CHK instruction word, or the ARGQ.

4.3 Tool Chain

The overall tool chain for deriving the hardware checks is illustrated in Figure 3. The process is fully automated; to the user, this process can be viewed as a simple compilation step. The tool chain accepts as input the application source code and produces two files:

1. The VHDL implementation of the application-specific error detectors. These are later synthesized for use in configuration of the FPGA device.
2. The application instrumented with CHK instructions is loaded into DRAM for execution on the main processor.

As previously mentioned, the application source code is all that is needed to derive and implement the detectors. We first compile the application source code using the augmented LLVM compiler presented in [13] which profiles the application to determine which variables are critical. The compiler then outputs the instrumented source code (to be run on the host processor) along with path and check definition files. The definition files contain all the information needed to implement the Path Tracking and Checking submodules and are fed into the *SDM VHDL Translator* which automatically generates their corresponding VHDL source code. Finally, we use Synplify Pro and Xilinx XST to generate a bit file used to program the FPGA device.

5 Evaluation

We have implemented the RSE and SDM alongside the Leon 3 processor [5] which includes a 7 stage in-order pipeline, split 16 KB L1 instruction and data caches, and a DDR memory controller. Synthesis and mapping were done with Synplify Pro 8.1 while translation and place & route were done with Xilinx XST 9.1. The target prototype board was a Digilent XUP board utilizing the Xilinx Virtex-II Pro 30 FPGA chip with a nominal clock speed of 65 MHz and 512MB of SDRAM.

We evaluated the SDM and associated hardware on seven applications from the Stanford Benchmark suite [7] according to the following metrics: (1) performance overhead and (2) area overhead of the synthesized design. We do not evaluate the coverage because it was thoroughly evaluated in [13] and is thus not the main focus of this study. Table 1 summarizes the number of checks which must be inserted into the source code for each benchmark in order to provide sufficient error coverage for the five most critical variables in each function.

Figure 4 shows the performance overhead attributed to both the software-only and hardware-supplemented implementations of the protected Stanford applications normalized against their un-instrumented original source. These results depict a marked improvement in performance across the applications when checking occurs within the SDM. However, the results also show that the performance benefit of CVR-based checking varies significantly between applications. After analyzing the instrumented source code, it is apparent that such variations likely occur as a result of *emitEdge* and *check* instructions which utilize the ARGQ as described in section 4. For example, some of the applications (e.g. Queens) require *emitEdge* instructions at commonly used branch points whereas other applications (e.g. Quicksort) do not.

Table 1. Number of checks needed for coverage of top 5 variables/function

Benchmark	Number of Checks	Number of CHK instruction activations
Bubblesort	5	748,502
IntMM	11	3,331,262
Oscar	15	175,920
Perm	16	2,047,922
Queens	10	2,525,002
Quicksort	14	135,630
RealMM	9	2,051,262

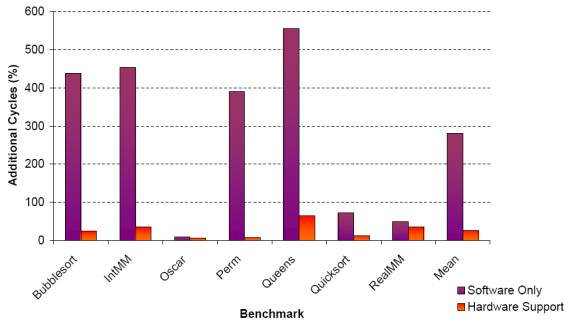

Figure 4. Performance Slowdowns for instrumenting the Stanford Benchmarks

Furthermore, investigation of the instrumented source code reveals that less than 1% of the performance overhead is actually attributable to the path tracking or recomputation hardware. Rather, CHK instructions embedded into the original source code are primarily responsible for the degraded performance. This occurs because insertion of CHK instructions, though they represent a no-operation from the point of view of the application, essentially add pipeline bubbles. Thus, applications like Queens, which require CHK instructions at commonly used branch points, require several additional CPU cycles.

Additionally, the combined impact of all such cycles is somewhat inflated on the Leon 3 platform due its low operating frequency and in-order pipeline. As the Leon 3 operates at 65 MHz, pipeline bubbles appear as costly as typically long-running operations (like memory transfers) which require far more CPU cycles when run on high frequency platforms such as the Pentium 4. Moreover, due to the in-order pipeline, these bubbles cannot be masked on the Leon 3 as would be possible on a superscalar architecture such as the Pentium 4. This is evidenced by the fact that, when run on a Pentium 4 chip, the software-only implementation of the instrumented Queens application required only 141% more CPU cycles to complete than the unmodified version; identical code when run on the Leon3 required 555% more cycles.

Figure 5 presents the additional area required by the SDM hardware as compared against the baseline Leon3. All area results were obtained by examining the equivalent gate count of the synthesized hardware. Also, synthesis results indicate that the SDM is not in the critical path of the design and thus has no effect on the clock frequency.

From Figure 5, it is clear that area overheads are dependent on the application to be protected. For example, although Quicksort has a smaller number of detectors than Perm, it has a higher area overhead (57% compared to Perm's 51.5%). This is because the checking expressions for Quicksort are more complex as they not only track the current path seen in execution, but also temporary state information gathered during recursive function calls. Perm, on the other hand, is not recursive and thus requires no information beyond the current path seen in execution in order to function.

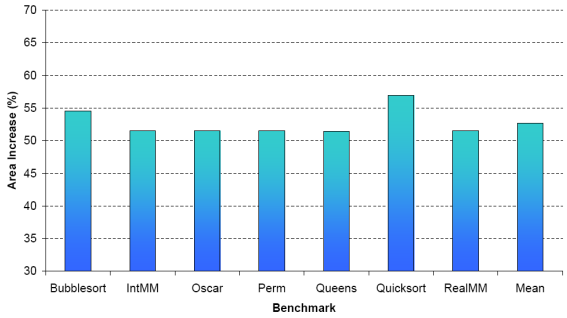

Figure 5. Area Overheads for the Stanford Benchmarks

6 Discussion and Conclusions

The SDM provides 75-80% coverage for errors that lead to program crashes as demonstrated in our software-only SDM implementation presented in [13]. Moreover, our implementation utilizes a practical detection strategy that avoids unnecessary benign error detection. The proposed tool chain automates the extraction of necessary checks and checking expressions while generating the hardware required to perform CVR. Our results show that, for the Stanford benchmarks, performance is only degraded by 27% while area is increased by 53% on average [2]. These results

[2]The gate count of the SDM hardware should not vary across different processors; area overhead would decline for more complex CPU designs

are encouraging as they demonstrate that error coverage approaching that of duplication is possible at a significantly reduced cost in terms of performance or area overheads.

In the future, we plan to pursue several improvements upon this implementation. For example, potentially unbounded area overheads possible with the current hardware implementation represents a point of concern. To this end, we have designed a specialized 51 instruction microcontroller capable of replacing the checking and path tracking hardware. Use of the microcontroller would also obviate the need to re-synthesize hardware for protected applications as its instructions can be stored in memory.

However, we do not want to rule out specialized hardware altogether as it is conceivable that some checking expressions may require too many instructions for the microcontroller to compute in a timely manner. Instead, we consider adding a feedback loop from the synthesis tools to the compiler in the tool chain. Thus, the compiler would "know" of any resource restrictions for the checks that it generates. For example, given resource limitations the synthesis tools could direct the compiler to generate either less complicated or fewer checks. On the other hand, the synthesis tools could direct the compiler to generate larger or more checks for increased coverage if resources are available.

Additionally, the area overhead of the specialized detectors could be improved by eliminating redundancies among checking expressions. For example, in the prototype implementation, combinational logic is not reused between expressions. If two expressions on two separate paths are identical, the same circuit is synthesized twice. Eliminating these redundancies could yield reductions in the area overhead for the error checking sub-module. However, reducing this redundancy will only be partially effective as much of the area overhead depends on hardware not specific to any particular checking expression (e.g. the StateStack).

Finally, as CHK instructions embedded at branch points diminish performance, we are examining methods which would reduce the need for such instructions. For example, information required by emitEdge CHK instructions could be inferred directly from signal taps on the main Leon3 pipeline. This method would not only reduce the observed performance overhead, but also reduce performance variations across different applications.

Once we have established these alternate checking implementations and extended the automated tool chain, we plan to examine the impact of the checking method across a broader variety of applications.

Acknowledgments

Special thanks to Flore Yuan for her excellent work in assuring correct operation of software-based checking on the Leon3. 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.

References

[1] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. *Electron Devices Meeting (IEDM)*, 2002.

[2] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proc. of the Int't Conference on Dependable Systems and Networks (DSN)*, 2000.

[3] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Translating ansi c to asynchronous circuits. In *10th Int't Symp. on Asynchronous Circuits and Systems (ASYNC '04)*, 2004.

[4] I. Corp. Intel core duo processor and intel core solo processor on 65 nm processor: Specification update, revision 017, 2008. http://download.intel.com/design/mobile/SPECUPDT/30922214.pdf.

[5] J. Gaisler. Leon 3 synthesizable processor. Online. http://www.gaisler.com.

[6] J. Gray. Why do computers stop and what can be done about it? In *Proc. of Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[7] J. Hennesey and P. Nye. Stanford integer and floating point benchmarks, 1988.

[8] R. K. Iyer, N. M. Nakka, Z. T. Kalbarcyzk, and S. Mitra. Recent advances and new avenues in hardware-level reliability support. *IEEE MICRO*, 25(6), 2005.

[9] I. Lee and R. K. Iyer. Software dependability in the tandem GUARDIAN system. *IEEE Trans. Softw. Eng.*, 21(5), 1995.

[10] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An architectural framework for providing reliability and security support. In *Proc. of the Int't Conference on Dependable Systems and Networks (DSN'04)*, 2004.

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

[12] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. Application-based metrics for strategic placement of detectors. In *Proc. of 11th Int't Symp. on Pacific Rim Dependable Computing (PRDC)*, 2005.

[13] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis, 2007.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. of the 3rd Int'l Symp. on Code Generation and Optimization*, 2005.

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

[16] M. Weiser. Program slicing. In *Proc. of the 5th international Conference on Software engineering (ICSE'81)*, 1981.

[17] X. Zhu and B. Lin. Hardware compilation for fpga-based configurable computing machines. In *DAC '99: Proc. of the 36th Conference on Design automation*, 1999.