Bare Demo of IEEEtran.cls for Conferences

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678-2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—The abstract goes here.**

I. Introduction

This demo file is intended to serve as a "starter file" for IEEE conference papers produced under LaTeX using IEEEtran.cls version 1.7 and later. I wish you the best of success.

mds

January 11, 2007

A. Background

Dual Modular Redundancy (DMR) is the traditional approach for detecting faults, where any difference between duplicate versions of operations on the same program state signals the precense of a fault. Typically this is performed with two identical hardware units executing in lock step and periodically comparing their outputs.

However redundancy is inherently costly consuming either power, or time, or often both, and is often not feasible in constrained systems such as satellites. For this reason a significant amount of research has been performed on attempting to reduce the amount of redundancy required to increase reliability. Due to the large body of work in this area this review will focus only on high-level approaches to reliability, examining current software techniques along with HLS techniques from the VLSI domain.

B. Generating Reliable Circuits using High Level Synthesis

Early VLSI work such as [**?**] examined how HLS tools could be used to reduce the amount of hardware required when duplicating the CDFG. The authors present a *Force-directed scheduling and alloction* algorithm, which aims to reduce the complete duplication of the resources by exploiting ones that are idle in each scheduling step.

Almost a decade later reliable HLS is revisited in [**?**], however instead of attempting to increase reliability through redundancy they increase reliability by using different versions of the components. They create a library characterisation of components which includes a reliability metric along with power, area, and latency. During the allocation phase of the HLS tool the library is searched to select the most reliable component while still meeting the other constraints. To achieve this a complete allocation, binding, and scheduling HLS algorithm is presented using ASAP scheduling. It starts by finding the most reliable solution, then it minimises the number of resources required, and finally finds a valid schedule that meets the designs latency requirement. If constraints cannot be met a victim node is selected and swapped out for a different less reliable version, and keeps repeating until a feasible solution is found.

Two years later Glaß et al expanded on [**?**] in [**?**] where instead of attempting to search the library of components for the most reliable combination, they present the result to the user as a design space exploration. Along with selecting different types of hardware units, this work also takes into account the replication of various components while still respecting other constraints. To perform the multi-objective design space exploration, an evolutionary algorithm is used however the runtime to search the space can be quite costly and is not always able to find feasible solutions.

Recently work In [**?**] extends [**?**] so that along with the library characterizations for reliability the length of time that a particular FU is active is taken into account. They argue that the length of time that the circuit is active will effect the reliability metric; a full allocation, binding, and scheduling is found using an ILP solver.

Other recent work includes [**?**] where the vulnerability of variables is analysed and the most vulnerable variables for a HLS program are protected by assigning them to special registers. The proposed variable vulnerability analysis takes into account their lifetimes, dependencies, and branch probabilities. Through analysing the CDFG, error and branch probabilities are used to propagate error likelihood forward to estimate the total vulnerability for each variable in the design. A Integer Linear Program solver is then used to minimise the subset of protected registers to cover the maximum number of variables. They build their results on top of LegUp, and use LLVM to perform the variable vulnerability analysis. They achieve impressive results, where protecting 20% of the registers results in more than 60% soft error mitigation in the best case. However full fault injection tests are not performed on the generated hardware, and only state-based probabilistic error testing was performed.

Summarising, previous work on reliable HLS has generally revolved around the idea of adding a reliability metric to component libraries and using that information to build more reliable circuits. However, these approaches make no guarantees about reliability and only aim to improve overall systems reliability without paying much attention to how the

reliability metric is derived. More recent work by Chen et al [**?**] is different in that a static analysis of the program is used to selectively duplicate sensitive variables.

C. Unreliable Hardware

Research into the use of unreliable hardware is slightly different to that of fault tolerance, since we can knowingly schedule instructions to execute on hardware that we don't trust. One tool that aims to aide the use of unreliable hardware is EnerJ [**?**] where type qualifiers are used to annotate whether a data type is approximate or precise. This tool takes an *all-or-nothing* approach to approximate computing, where precise annotated types are computed exactly, but imprecise ones have zero guarantees. Strict constraints on how data flow occurs insure that the precise and approximate code regions are segregated; with approximate-to-precise data flow illegal, and precise-to-approximate legal. The authors provide a Java extension for the annotated type system, and many examples of code that can tolerate approximate regions demonstrating energy savings between 10-50%.

RAISE [**?**] is a recent project that aims to schedule instructions taking into account temporal and spatial vulnerabilities. They call the most likely time an instruction will receive and error the vulnerable period and this is calculated for every instruction. The tool then adjusts the instruction schedule so that the most vulnerable instructions have the least vulnerable period, minimising the overall likelihood of an error in the system. An example of this is when a multiply operation causes a pipeline stall which in turn causes a *load* or a *store* operation to get stuck at the decode stage of the processor pipline; the longer this operation is stuck in the decode stage, the more likely it's opcode or address could be corrupted.

A metric called the Instruction Criticality Factor (ICF) is determined for every instruction through analysing the total number of dependent instructions that would be effected if it experienced a fault. This metric is then used to rearrange the execution order of the most critical instructions in order to increase the overall reliability of the program.

Summarising, the unreliable hardware field has also started looking towards high-level approaches to managing imprecise compute units. Either through a type system where users propose precise and imprecise regions of code, or through analysis of code and low level architecture models to influence scheduling decisions.

D. Software Soft-Error Protection

The hardware expense required for reliability has provided motivation for research into software fault tolerance. Many papers argue that with shrinking feature sizes and increasing use in power saving techniques such as DVFS reliability will be a concern for all processor markets, such as desktops, which might not be able to afford double or triple redundant execution cores [**?**].

Some software fault tolerance research requires modification to the underlying architecture especially when making use of multicore and multithreaded processors. One example of this is SRTR [**?**], simultaneously and redundantly threaded processors with recovery, which achieved fault tolerance through using two simultaneously running identical threads. However instead of running both threads in lock step one of them is executed slightly before the other. The trailing thread is responsible for checking that the execution of the leading thread is valid and completed before the leading thread commits it's results. To achieve this a new architecture is proposed which uses queued access to the register file in for checking results and to prevent it from becoming a bottle neck.

In an attempt to reduce architecture dependence research into software only fault tolerance techniques became popular in the mid 2000's. Early notable work include EDDI (Error Detection by Duplicating Instructions) by Oh et al [**?**] which is a compiler tool for super-scalar processors that attempts to duplicate instructions using different registers and variables. It tries to schedule operations to make use of instruction level parallelism while still preserving its fault tolerant properties.

In an attempt to further reduce the overhead introduced by software fault tolerance Oh et al, [**?**] used signatures to ensure the correctness of the control flow graph (CFG) of a program. A static signature is assigned to every block and compared against a runtime signature during execution, flagging an error if there is a mismatch. This ensures that a transition between two basic blocks after a branch operation is a valid one according to the CFG. This method protects against transient faults in the program counter, and is often used as a method to detect malicious activity trying to alter the execution path of a particular program.

SWIFT [**?**], is another popular software only fault tolerance tool that aims to combine the methods presented in [**?**] and [**?**] for a single threaded application. Through transforming the code it is able to reclaim unused instruction level resources for duplicating instructions and inserting comparison points to check the replications. By combining these various techniques SWIFT is able to achieve a 14% speedup to other single threaded techniques which can achieve full fault tolerance coverage.

Summarising, software fault tolerance techniques tend to either fall into two broad categories; techniques that require small architecture changes, such as modifying the pipeline to allow efficient checking between multiple redundant threads [**?**]; and techniques that are purely software requiring no architecture changes. In a similar fashion to reliable HLS various approaches have been modified to trade off reliability and fault coverage for overhead. Compiler techniques such as static analysis of the code play an important role, especially in the scheduling of redundant operations and in deciding which portions of the program should be protected.

II. Conclusion

The conclusion goes here.