Processor-level Selective Replication

Nithin M. Nakka, Karthik Pattabiraman, Zbigniew T. Kalbarczyk, Ravishanker K. Iyer

***Abstract*—Even though replication has been widely used in providing fault tolerance, the underlying hardware is unaware of the application executing on it. The application cannot choose to use redundancy for a specific code section and run in a normal, unreplicated mode for the rest of the code. In this paper we propose *Processor-level Selective Replication*, a mechanism to dynamically configure the degree of instruction-level replication according to the applications demands. The application can choose to replicate only code sections that are critical to its crash-free execution. This decreases the impact on the performance. It is also known that many of the processor-level faults do not lead to failures observable in the application outcome. So, selective replication also decreases the number of false positives.**

***Index Terms*—Replication, Reconfiguration, Critical Code Sections**

I. Introduction and Motivation

Replication, as a fault-tolerance technique, has been widely at the application, processor and the system levels. Replication can be introduced into the application at compile time by duplicating the instructions in the static source code and providing code for comparing the outputs of the duplicated instructions [1]. This has the advantage that the underlying hardware does not need to be modified. The drawback of this approach is that it incurs a high memory and performance overhead.

The two basic approaches for processor-level replication are: *hardware redundancy* and *time redundancy*. 1) *Hardware Redundancy* [2] is achieved by carrying out the same computation on multiple, independent hardware at the same time and corroborating the redundant results to expose errors. 2) *Time redundancy* [3][4] obtains redundant computation by repeating the same operation multiple times on the same or idle hardware. It has lower performance overhead than the software-implemented replication but much higher performance overhead than hardware redundancy.

In either type of redundancy the underlying hardware is unaware of the application executing on it. The application cannot choose to use redundancy for a specific code section and run in a normal, unreplicated mode for the rest of the code.

Recent work [e.g., [5][11]] has shown that it is feasible to identify some critical variables in an application, which when in error will cause system/application failure with a high probability. Protecting the computation of these variables can provide a high coverage against program failures (crashes[1] and fail silence violations). This motivates replication of only those portions of the application that compute the critical variables, instead of replicating the entire application. To enable this at a low performance overhead, replication at the processor-level needs to adapt to application needs.

We propose hardware-based selective replication which enables the application to choose which portions need to be replicated and what would be the degree of replication. The application is instrumented at compile-time with special CHECK instructions, an extension to the instruction set architecture (ISA). These CHECK instructions invoke a reconfiguration of the underlying hardware to provide the specified level of replication.

This paper addresses the following two questions to provide selective replication:

- Which sections of the code need to be replicated and to what degree?
- How can we modify the renaming, issue and commit mechanism to handle a specified level of redundancy for portions of the code?

II. What to replicate and how much?

Recent experiments by Pattabiraman et. al. [5], have shown that we can identify program variables, which when in error lead to a program crash or hang with a high probability. These variables will be referred to as *critical* variables. Using program *fanout* as a metric the location and variable that needs to be checked can be identified. It has been demonstrated that with small number of ideal detectors relatively high coverage (60% for crash detection in *gcc*) is achievable. The claim of this paper, therefore, is that if the computations of the critical variables can be replicated then this can enhance application dependability very substantially for a small performance overhead compared to full replication.

[1] Our aim is to preemptively detect program crashes as they are not always benign [12]

A. Procedure for extraction of Critical Code sections

Any part of the application that affects the value of a critical variable is a critical code section (consisting of *critical* instructions). Any code section includes:

- Instructions that define *critical* variables.
- Instructions that produce a result that is subsequently consumed by *critical* instructions.

We use a *reverse depth-first search* algorithm for automated identification/extraction of instructions that directly or indirectly affect the value of critical variables. The pseudo code (in *python*) implementing the algorithm is shown in Fig. 1.

```
Dependencies(Metric):
  set value(node) {
    "Actual computation of dependencies"
      if ( node.ID) in visited:                (1)
        return data[ int(node.ID) ]            (2)
      dependSet = set([(node.PC)])             (3)
      visited.add( node.ID )                   (4)
      for (predNode) in node.Preds():          (5)
        predSet = value( predNode )            (6)
        dependSet = dependSet.union(predSet)   (7)
      numNodes += 1
      data[ node.ID ] = dependSet              (8)
      return data[ node.ID ]                   (9)
```

Fig. 1. Pseudo-code for extracting critical code sections

`value()` accepts a dynamic instruction as its argument and returns the set of instructions on which the given instruction is directly or indirectly dependent. To extract the critical code sections we invoke `value()` on all the instructions that define the critical variables. The identified critical code section can be replicated. An important point to note is that the when using multiple critical nodes, there may be an overlap in the instructions that affect two or more nodes. We take advantage of the optimization that all such instructions which affect multiple critical nodes need to replicated only once for all nodes, instead of being replicated for each node.

III. Hardware implementation

In order to support selective replication, we need to modify the control logic that updates the register alias table (RAT) and that which commits the instructions from the ROB (commit control logic). The *register alias table* (RAT) is used in register renaming. It contains as many entries as the number of architectural registers. The i^{th} entry in the RAT contains information of the source of the most recent value of register *i*. The ROB contains an entry for each in-flight instruction in the pipeline. It holds the information about the source and destination operands of the instruction.

The ROB is augmented with a bit (referred to as the REPL bit) to indicate whether it contains a replicated or an unreplicated instruction. ROB designs are of two types. One in which the result of the instruction in the ROB entry is written to separate physical register file, and the other in which the result is written to the ROB entry itself. We assume the ROB where the results are written to the ROB entry itself. The RAT and commit control logic for the unreplicated mode is the same as that used in the normal superscalar out-of-order pipelines.

Let *a*, *b*, *c* … be the instructions in the replicated code segment. For any instruction *i* in this segment let $i_1, i_2…, i_r$ be the *r* copies of the instruction. The *r* copies of instruction *a* ($a_1, a_2…, a_r$) are allocated the first unallocated consecutive entries in the ROB following by the *r* copies of instruction *b* and so on. If the ROB has less than *r* free entries, none of the copies of a replicated instruction are dispatched. This ensures atomic dispatch of all copies of a replicated instruction and facilitates updating the RAT with the dependencies for all the copies of the instruction. The ROB capacity is reduced by a factor of *r*.

1) Register Renaming: If a replicated instruction *d* reads from register *$x*, the RAT entry for *$x* is looked up. If the value of *$x* is available in the architectural register file then all copies of *d* get the value for this source operand from the architectural register file. Otherwise, the value of *$x* is the result of an in-flight instruction, *p,* that is allocated the ROB entry *k.*

If *p* is an unreplicated instruction (as indicated by the REPL bit in entry *k*) for all replicas $d_1, d_2, …, d_r$ the source operand register is renamed to read from entry *k*. If *p* is a replicated instruction the register operand *$x* of d_i is renamed to read the output from instruction p_i, where *i*= 1, 2, 3…, *r*.

2) Issuing and executing instructions: With the above renaming mechanism the issue and execution of instructions to functional units can be done without any modification to the already existing scheduling mechanism. After an instruction has completed execution in the functional unit, the result is stored in the ROB entry corresponding to that instruction itself. For memory access instructions, the result of the address generation is stored in the ROB entry.

3) Committing Instructions: Each ROB entry contains a field to indicate if the instruction is ready to commit or not. Committing unreplicated instructions follows the same procedure as committing an instruction in a pipeline without support for replication.

Among replicated instructions two classes of instructions, memory access instructions and the rest, are treated separately. When a replicated instruction memory access at the head of the ROB has completed execution (generated effective address), all of its copies are checked to see if they have completed execution. If not, the action is postponed to the next cycle. If all *r* copies have generated their effective addresses (which is stored in the result field of the ROB entry), these results are compared against each other. If there is a mismatch an error is raised and appropriate recovery action is taken. If the effective addresses of all *r* copies match then a single memory access request is sent to the memory subsystem, on behalf of all the replicas. This reduces the pressure on the memory bandwidth, but loses the coverage over possible errors in memory access. When this memory access is complete, all copies of the instruction are ready to commit. In case of a load the data read is written to the architectural register file. The entries from the ROB and the LSQ for all copies are de-allocated. When any other replicated instruction is at the head of the ROB, all of its copies are checked to see if they are ready to commit. If all *r* copies are ready to commit the result fields in their ROB entries are compared to verify the computation. If all *r* fields match, the instruction is committed and the result is committed to the architectural register file.

IV. Experimental methodology

The sim-outorder, cycle-accurate performance simulator from the SimpleScalar Tool Set [8] has been augmented to provide selective replication. The MIPS-based, PISA instruction set has been extended to include the CHECK instruction to invoke replication from within the application stream, based on application needs. An important aspect of this work is the determination of the error coverage of the proposed approach (rarely provided by the usual architecture-level studies of replication). The error coverage provided by selective replication is measured along with the performance overhead incurred.

For error coverage analysis the *print_tokens* benchmark from the Siemens Suite [9] was used. The relatively short execution times of these benchmarks facilitate conducting large number of fault injection experiments so as to provide statistically significant results. *print_tokens* breaks the input stream into a series of lexical tokens according to pre-specified rules. The benchmark was statically analyzed and critical variables and code sections were identified and replicated. Random faults were injected into dynamic instructions and data values and the error coverage is measured.

A. Fault Model

The following fault models are considered to evaluate the coverage provided by selective replication:

1) Instruction Errors: Errors in instruction binary while the instruction is being executed in the pipeline. These errors can occur during the transfer of the instruction from the cache to the pipeline or while the instruction is being decoded in the pipeline.

2) Data Errors: Errors in the output of a functional unit that may be written to a register or used as an effective address for a memory access instruction. ECC in memory, cache, or registers does not protect against these errors. This is because the correct ECC would be calculated on the wrong data and would be written to registers.

B. Results for Error Coverage

Based on the *fanout* metric (see discussion in Section II.A) three variables with the highest *fanout* were chosen. The critical code sections affecting the chosen variables were extracted. There are 586 instructions in the critical code section and the application incurs an overhead of 7.4% to replicate these critical instructions. Full duplication incurs an overhead of 15.2%.

The outcomes of the fault injection experiments are classified into the following categories:

Not Activated (NA): The instruction was not executed, or the instance of data was not calculated by a functional unit.

Not Manifested (NM): The error did not produce any observable effect on the application or the system.

System Detection (SYS): The application crashed due to an exception.

SELREP Detection (SELREP): The error was detected by the selective replication mechanism.

Fail Silence Violation (FSV): The application completed execution but produced incorrect results.

Program Hang (HANG): The program was hung.

TABLE I shows the results for injection of transient faults in data at the output of functional unit. TABLE II shows the results for injection of transient faults in the binary of an instruction in the dynamic instruction stream.

TABLE I
Results of transient fault injection in output of func. units for *print_tokens*

Outcome	Baseline (No Duplication)	Full Duplication	Selective Replication
# faults	971	971	971
% Replicated Insns	0	100%	7.9%
NA	8.8%	0.0%	0.0%
NM	62.9%	0.0%	40.4%
SYS	10.8%	0%	2.5%
SELREP	0.0%	100%	67.5%
FSV	79.3%	0.0%	26.5%
HANG	9.9%	0.0%	3.5%

TABLE II
Results of transient fault injection in binary of instructions for *print_tokens*

Outcome	Full Duplication	Selective Replication
# faults	997	997
% Replicated Insns	100%	7.9%
NA	0.0%	0.1%
NM	22.5%	42.7%
SYS	10.5%	21.2%
SELREP	88.4%	51.5%
FSV	0.9%	21.0%
HANG	0.3%	6.3%

The results in

TABLE I show that for data errors selective replication provides coverage of 67.5%. In contrast, full duplication covers almost all the errors. As shown in TABLE II, for instruction errors selective replication has a lower coverage of about 51.5%. Full Duplication covers 88.7% of the instruction errors. A very important drawback of using full replication is that it detects many of the errors that were not manifested in the baseline case. From the application perspective, these are false positives and are undesirable.

References

[1] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Error Detection by Duplicated Instructions in Super-Scalar Processors,” IEEE Trans. Reliability, Mar. 2002, pp. 63-75.

[2] C. Weaver, T. Austin. “A Fault Tolerant Approach to Microprocessor Design,” in Proc. of the International Conference on Dependable Systems and Networks, pp. 411-420, July 2001.

[3] T. Vijaykumar, I. Pomeranz, K. Cheng. “Transient Fault Recovery Using Simultaneous Multithreading,” Proc. 29th Annual Int'l Symp. on Computer Architecture, May 2002.

[4] Joydeep Ray, James C. Hoe, Babak Falsafi. “Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery,” in Proc. 34th Annual Int’l Symposium on Microarchitecture, Austin, Texas, pp. 214-224, Dec 2001.

[5] K. Pattabiraman, Z. T. Kalbarczyk, and R. K. Iyer. “Application-Based Metrics for Strategic Placement of Detectors,” To appear in the Proc. of Pacific Rim Dependability Conference 2005.

[6] Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[7] R. Chillarege, W-L Kao, and R. Condit, “Defect Type and its Impact on the Growth Curve,” Proc. 13th Intl. Conference on Software Engineering, 1991.

[8] D. Burger, T. Austin, and S. Bennett, Evaluating Future Microprocessors: The SimpleScalar ToolSet, University of Wisconsin-Madison, Computer Sciences Department, Technical Report CS-TR-1308, July 1996.

[9] M. Hiller, A. Jhumka, and N. Suri, On the Placement of Software Mechanisms for Detection of Data Errors, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2002.

[10] R. K. Iyer, N. Nakka, Z. T. Kalbarczyk, S. Mitra. Recent Advances and New Avenues in Hardware-Level Reliability Support, To appear in IEEE MICRO Nov/Dec 2005.

[11] M. D. Ernst, *Dynamically Detecting Likely Program Invariants*, PhD Dissertation, University of Washington, Department of Computer Science and Engineering, August 2000.

[12] C. Basile, L. Wang, Z. Kalbarczyk and R.K. Iyer, "Group Communication Protocols under Errors," *Proc. 22nd Symposium on Reliable Distributed Systems, SRDS'03*, Florence, Italy, 2003.