Processor-level Selective Replication

Nithin Nakka, Karthik Pattabiraman, Ravishanker Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory

{nakka, pattabir, iyer}@crhc.uiuc.edu

Abstract

Full duplication of an entire application (through spatial or temporal redundancy) would detect many errors that are benign to the application from the perspective of the end-user. It has also been seen that duplication has upto 30% performance overhead and needs significant introduction of hardware to synchronize the replicas. In order to overcome the drawbacks of performance overhead and detection of "benign" faults, we propose a processor-level technique called Selective Replication, which provides the application the capability to choose where in its application stream and to what degree it requires replication. Recent work on static analysis and fault-injection based experiments on applications reveals that certain variables in the application are critical to its crash- and hang-free execution. If it can be ensured that the computation of these variables is error-free, then a high degree of crash/hang coverage can be achieved at a low performance overhead to the application. The Selective Replication technique provides an ideal platform for validating this claim. The technique is compared against complete duplication as provided in current architectural-level techniques. The results show that with about 59% less overhead than full duplication selective replication detects 97% of the data errors and 87% of the instruction errors that were covered by full duplication. It also reduces the detection of errors benign to the final outcome of the application by 17.8% as compared to full duplication.

1 Introduction

System level replication has been a widely used technique to detect and possibly tolerate transient errors in both commercial and research prototypes. Processor-level replication has also been used recently [1]. 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 [2]. This has the advantage that the underlying hardware does not need to be modified. Additionally, using compiler analysis techniques, only critical portions of the application can be chosen to be replicated, instead of the entire application. 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 [3] – carrying out the same computation on multiple, independent hardware at the same time and comparing the redundant results. (2) Time redundancy [4][6] –executing the same operation multiple times on the same or idle hardware. In either type of redundancy, the underlying hardware is unaware of the application executing on it. All instructions of the application are replicated and checked for correct execution. 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 other words, it is a "*one size fits all*" approach.

Another advantage of selectively replicating an application is the reduction in detection of processor-level errors that do not affect the final outcome of the application. Fault-injection based experiments by Choi [7], Wang [8] and Saggese [9] showed that 80%-85% of the errors did not manifest as errors in the application outcome. Full replication at the hardware level aims at detecting all errors in the processor, even those that are benign to correct application outcomes. This leads to false alarms to the operator, which are considered undesirable from a safety perspective.

We propose hardware-based selective replication to achieve the advantages of both software- and hardware-implemented replication. The application can choose which portions need to be replicated and the degree of redundancy. This is achieved by compile-time instrumentation of the application with special CHECK instructions, an extension to the instruction set architecture (ISA) to invoke a reconfiguration of the underlying hardware and provide the specified level of replication.

Recent work by Pattabiraman et al. [7][11] has shown that it is feasible to identify some critical variables in an application, which when in error will cause application/system failure with a high probability. Based on this study it was concluded that protecting the computation of these variables can provide a high

coverage against program failures (crashes[1] and fail silence violations). Selective replication provides a platform to validate this claim because it allows replication of *only those portions of the application that compute the critical variables*, instead of replicating the entire application.

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

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

The mechanism of replication in a superscalar processor has been detailed and a possible implementation is presented. The results show that even though Selective Replication detects about 87% of the instruction errors and 97% of the data errors detected by Full Duplication, it incurs only 59% overhead. Moreover, the detection of errors benign to the application outcome is reduced by about 18%.

2 What to replicate

In order to identify the *critical* variables we use the approach similar to that described in [7]. The criticality of the variables to error-free execution of the program has been evaluated using metrics like *lifetime*, and *fanout* (definition). It was shown that ideal detectors placed at locations with high fanout gave higher coverage, where an ideal detector is one that is able to detect any data error that propagates to the location at which it is placed. The analysis was done on the program's dynamic dependency graph (DDG). For multiple inputs, faults are injected into the points that are being evaluated for criticality (with high fanout, lifetime etc.). For each input, the effect of each fault is traced, using the DDG for that input, to locations of the program where the program may crash. If the error led the program to a potential crash location, a detector at the critical point can detect an impending program crash. The claim of this work 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. Moreover, the types of errors detected are, for the most part, those that would need full duplication or extensive coding to detect.

Extraction of the 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 critical code section includes:

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

A *reverse depth-first search* algorithm is used for automated identification/extraction of instructions that directly or indirectly affect the value of critical variables. Using selective replication *only* these critical instructions in the program are replicated whereas all other instructions are executed normally. Due to a constraint of space, we do not present the details of the reverse depth-first search algorithm here, but are presented in [24]. An important point to note is that when using multiple critical nodes, there may be an overlap in the instructions that affect two or more nodes. All such instructions that affect multiple critical nodes need to be replicated only once for all nodes, instead of being replicated for each node.

In summary, the backward slice of the instruction that defines a critical variable for selective replication is extracted. Backward slicing in a static program segment is known to be very time-consuming. Using a dynamic dependency graph the execution time for backward slicing is reduced. In addition, calls to library functions have not been traced; rather, the entire library function is considered a single node in the dynamic dependency graph.

Formally, let $\boldsymbol{\Theta}$ be the set of critical variables, and $\boldsymbol{I}$ be the set of all inputs. For an input $i \in \boldsymbol{I}$, let the dynamic dependency graph be $\boldsymbol{G_i} = (\boldsymbol{V_i}, \boldsymbol{E_i})$ where the vertices in $\boldsymbol{V_i}$ correspond to statements in the dynamic execution of the program and there is an edge (u, v) in $\boldsymbol{E_i}$ if statement u is executed before statement v and u produces a result that is used by v.

For every critical variable $\theta \in \Theta$, let $\boldsymbol{H}_{i,\theta} = (\boldsymbol{V}_{i,\theta}, \boldsymbol{E}_{i,\theta})$ be the subgraph of $\boldsymbol{G_i}$ which is the backward slice of instructions that affect variable θ. For each dynamic instruction $w \in \boldsymbol{V}_{i,\theta}$ its counterpart, s, in the static code segment is found (both of them have the same PC). The set of static instructions corresponding to the dynamic instructions in $\boldsymbol{V}_{i,\theta}$ is the set of critical instructions, $\boldsymbol{S}_{i,\theta}$, that need to be replicated for input i and critical variable θ. $\boldsymbol{S}_i = \bigcup_{\theta \in \Theta} \boldsymbol{S}_{i,\theta}$ is the set of critical instructions for input i and $\boldsymbol{S} = \bigcup_{i \in I} \boldsymbol{S}_i$ is the set of all instructions in the static code segment that need to be replicated for all inputs considered.

In a real implementation a compiler places a special CHECK instruction before and after each duplicated instruction to notify the hardware of the change in level of replication (in our simulation we insert the check instruction into the dynamic instruction stream

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

when the first replicated instruction and after the last replicated instruction are encoutered). Note that the critical instructions can also be consequent to each other. In such a case, for each block of contiguous critical instructions, one CHECK instruction is placed before and one after the block of instructions to notify the replication module of ENTERING into and EXITING from replication mode.

3 Overview of Selective replication

This section describes the selective replication technique in detail and presents a possible hardware implementation in a superscalar processor. Instructions are fetched as in a normal pipeline. The dispatch mechanism, which allocates reorder buffer entries to the currently fetched instructions, broadly operates in two modes: the unreplicated mode and the replicated mode. In the unreplicated mode, a single copy of each instruction is dispatched, renamed, and allocated to the reorder buffer (ROB). In the replicated mode, r copies of each instruction are dispatched, where r is the degree of replication. If any instruction, i, in the replicated code consumes a value produced by a preceding unreplicated instruction, j, then all copies of i receive their input from j. If a replicated instruction i_1 is dependent on another replicated instruction i_2, then the copy of i_1 in every replica is dependent on the copy of i_2 in the same replica. Thus, the register operands of the instructions are renamed.

After instruction execution is complete the result is stored in the ROB itself. When an instruction at the head of the ROB is ready to commit, all copies of the instruction are checked to see if they are ready to commit. If all copies are ready to commit, then their results (stored in their corresponding ROB entries) are compared. If all of them match the instruction is committed. In the case of even a single mismatch appropriate recovery action can be taken.

3.1 Mechanism of replication

An important aspect of this work is the mechanism for selective replication that allows the application to choose the extent and location of replication it needs. In this section the implementation of selective replication in a modern superscalar out-of-order processor is described. Implementing selective replication in a superscalar processor involves modifying the instruction fetch and dispatch, register renaming, and commit mechanisms of the processor. The block diagram in Figure 1 shows a processor pipeline (top of the figure) with the modifications required for selective replication (bottom of the figure).

Before describing the actual mechanism of execution in the replicated mode, it is helpful to describe some key hardware data structures that would be used in the execution. The *register alias table* (RAT) is used in dynamic scheduling in the rename state of pipeline. 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. If the most recent instruction producing register i has been committed to architectural state, the i^{th} entry in the RAT contains a special sentinel value indicating that the value of a register is ready and available in the architectural register file. If the most recent instruction producing register i is still executing and is in the ROB, the entry in the RAT contains the index of the ROB entry containing the instruction. Thus the RAT holds information of the (*read-after-write*) RAW dependencies among instructions.

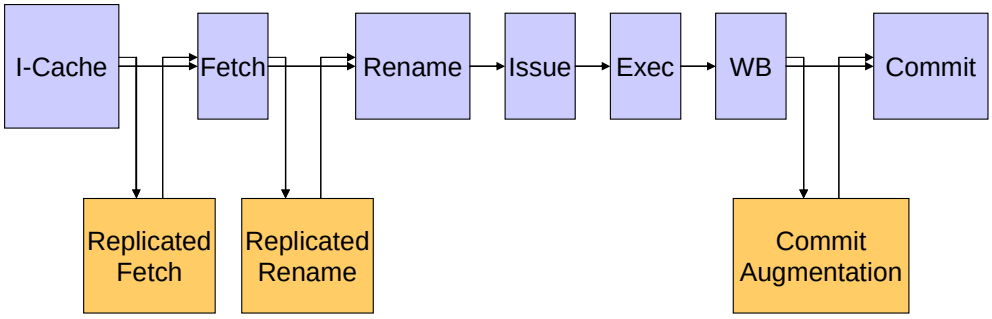

Figure 1: Modifications to pipeline for selective replication

The load/store queue (LSQ) contains entries for all the memory access instructions (loads and stores) that are currently in-flight. The LSQ can be used to optimize loads by forwarding the data from the immediately previous store, if both generate the same effective address and are writing the same number of bytes.

The replicated fetch mechanism shown in Figure 1 provides multiple copies of a fetched instruction to the dispatcher. The detailed hardware implementation of this mechanism is presented in Section 4.

Replicated Rename. The mechanism for renaming multiple copies of an instruction, based on the replica index, is shown in Figure 4. 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*.

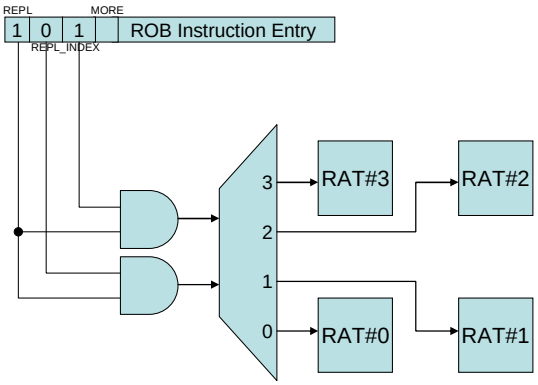

Figure 2: Mechanism for register renaming of multiple instructions

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.

Instructions issue to functional units. With the above renaming mechanism the issue of instructions to functional units can be done without any modification to the already existing scheduling mechanism.

Execution and Storing the Result. The instructions in the unreplicated mode are always executed in a noral out-of-order fashion. The instructions in the replicated mode also execute in an out-of-order fashion, though it complicates the mechanism to detect the completion of all copies of the instruction. It provides the benefits of superscalar out-of-order execution by exploiting the instruction level parallelism and increasing the utilization of the multiple functional units available for instruction execution.

The ROB need not be empty before switching from the unreplicated mode to the replicated mode. This can be done by maintaining the information in the register alias table across the two modes. In other words, if one of the replicated instructions reads from a register which is produced by a previous unreplicated instruction (which is not committed and still holds an entry in the ROB), then all copies of the replicated instruction read from the result of the same unreplicated instruction. For dependencies among instructions within the replica, a replicated instruction that is dependent on another replicated instruction gets its input from the producing instruction in the same replica.

For switching from the replicated mode to the unreplicated mode, however, the constraint that the ROB is empty before the switch is maintained. This is because an unreplicated instruction *i*, that is dependent on an instruction *j* in the preceding replicated code, is effectively dependent on all the copies of *j*. Before issuing *i*, all copies of *j* must have completed execution and their results matched so as to forward the result to instruction *i*.

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.

Commit Augmentation. As shown in Figure 1, the commit unit is augmented to vote on the results of multiple replicas to support selective replication. The commit stage is augmented to vote on the 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 memory access instruction 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 commit 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 deallocated. 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.

4 Hardware implementation

The mechanism to dispatch multiple copies of instructions is depicted in Figure 3. Instructions are fetched into a temporary fetch buffer (temp_fetch_buf in Figure 3). Depending on the degree of replication different number of copies of the instruction should be dispatched. In a processor that does not support replication the input to the dispatch mechanism would be the instructions in the temporary fetch buffer. These instructions are dispatched to the reorder buffer based on the space available in the ROB and the dispatch width (the maximum number of instructions that can be dispatched in one clock cycle) of the processor.

In the processor with selective replication depending on the degree of replication requested by the application, the instructions that are dispatched in the current clock cycle need to be determined. The replicated instructions that can be dispatched in the current clock cycle are placed in the real fetch buffer (fetch_buf in Figure 3). The degree of replication is stored in the register REP_LVL. REP_LVL – 1 (calculated using the adder shown in Figure 3) is used as an index into the combinational logic that starts with 0 when there is no replication. Consider a processor with

- fetch width, max. number of instructions fetched in a clock cycle = 4 and
- dispatch width, max. number of instructions dispatched in a clock cycle = 4

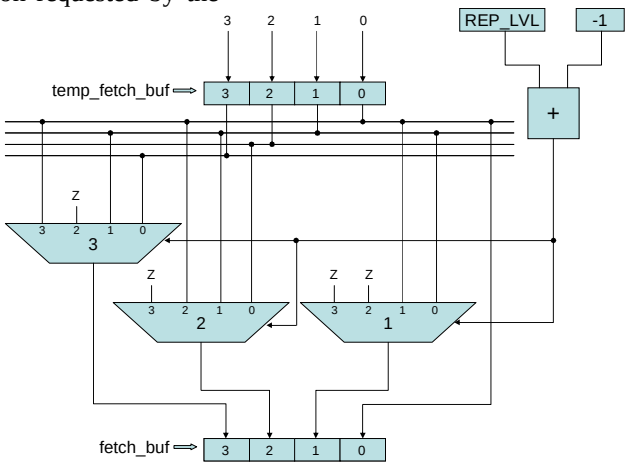

Figure 3: Mechanism for dispatching multiple copies of instructions

Based on the value of REP_LVL, the entries in the temp_fetch_buf are stored into the fetch_buf as shown in Figure 3. The mechanism implements the following rules:

- If REP_LVL = 1 (no replication) all entries in temp_fetch_buf are copied to corresponding entries in fetch_buf.
- If REP_LVL = 2 (duplication) two copies of the instructions in entry 0 are passed to the entries 0 and 1 in fetch_buf and two copies of entry 1 in temp_fetch_buf are passed to the entries 2 and 3 in fetch_buf. Only the entries 0 and 1 in the temp_fetch_buf are invalidated and there is space in temp_fetch_buf for 2 more instructions to be fetched in the next clock cycle.

This case is depicted in Figure 4. The red lines show the duplicated instructions being routed from the temp_fetch_buf to the fetch_buf. The cases for REPL_LVL = 3 and 4 are similarly defined.

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. The replication mechanism is presented assuming an ROB design where the results are written to the ROB entry itself, though it is possible to extend the technique for the design where a separate physical register file is used to store the results of instructions. The additional hardware required in the context of RISC architecture is described (A RISC architecture uses load/store instructions to access memory and arithmetic instructions whose destination is a register). The RAT and commit control logic for the unreplicated mode is the same as that used in the normal superscalar out-of-order pipelines.

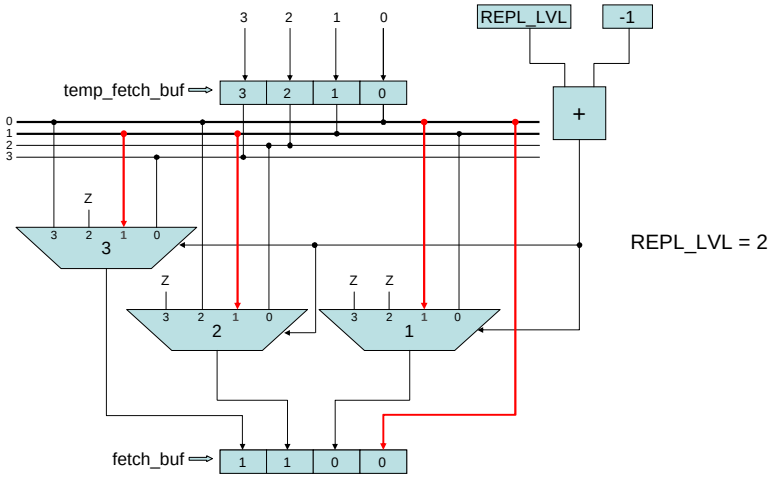

Figure 4: Example of selective replication with a replication level of 2

5 Evaluation Methodology

The software-implemented functional simulator implements a MIPS-based SuperScalar processor. The *sim-outorder* processor performance simulator of the SimpleScalar Tool Set [14] has been augmented to simulate the RSE with embedded hardware modules. *sim-outorder* simulates an out-of-order pipelined processor. The main loop of the simulator is executed once for each target (simulated) machine cycle. Currently, CHECK instructions are embedded at runtime, and not at compile time, as mentioned in Section 2. At the time of an instruction fetch, the simulator determines whether to insert a CHECK instruction before it into the instruction stream. It does this either by decoding the instruction or by monitoring the fetched instruction address. This is equivalent to the CHECK instruction being embedded in the static instruction stream of the program.

5.1 Workload for Evaluation

Evaluation of the performance overhead and error coverage is based on the Siemens suite of benchmarks. These benchmark applications are representative of real-world programs and contain a few hundred lines of code [15]. They provide a rich input set with an average of 3400 inputs for each benchmark. For each benchmark, we choose the first 100 inputs from its input set. For each input *i* the dynamic dependency graph, G_i of the program is generated. For each critical variable, its backward slice in G_i is calculated. The set of nodes (instructions) in the backward slice are critical instructions that need to be replicated. In a similar manner, the set of critical instructions for each critical variable are extracted. The union of these different sets of critical instructions is calculated. This procedure is repeated for each input *i* in the chosen set of 100 inputs. The set of critical instructions that is replicated is the union of the sets of critical instructions for all the inputs. Table 1 gives a brief description of the Siemens suite of benchmarks.

Table 1: Siemens suite of benchmarks

Benchmark	#loc	Description
schedule	412	A priority scheduler for multiple job tasks. Given a list of tasks finds an optimal schedule
schedule2	373	Same operation as Schedule but a different implementation.
print_tokens	727	Breaks the input stream into a series of lexical tokens according to prespecified rules.
print_tokens2	569	Using the tokenizer interface Breaks the input stream into a series of lexical tokens according to prespecified rules.

5.2 Performance Overhead

The software-level implementation evaluates the performance overhead incurred by the framework and modules in terms of additional processor cycles.

5.2.1 Overhead categories

The experiments evaluate the following two kinds of overheads:

1. *Framework Overhead.* This is the overhead incurred by the processor due to the presence of the framework without any modules instantiated. In such a case, the framework does not perform any checking and is decoupled from the pipeline. The overhead incurred in the performance of the application is due to the *memory arbiter* introduced to arbitrate memory accesses of the processor and the RSE.
2. *Performance Overhead Due to Selective Replication.* The performance overhead incurred by the application is measured in terms of the number of additional

cycles taken to execute the application in comparison to the baseline processor (without the framework). In order to show the need for selective replication, a randomized replication mechanism (RANDOMREP) is also evaluated where instructions are randomly replicated. So as to make a fair comparison between the randomized and selective replication approaches the fraction of replicated instructions is maintained, ensuring that the overheads are approximately equal.

There can be other sources of overhead due to the additional hardware introduced in the processor. The additional circuitry will increase the capacitive load on pipeline nodes. This will in turn lead to a slight increase in the clock cycle time. Because we are doing a functional simulation this factor of overhead is not included in our experiments.

5.2.2 Results

Table 2 and Figure 5 show the overheads incurred, for different applications, due to the framework with selective replication (SELREP) in comparison with full replication (FULLREP). The overhead of the framework with no modules instantiated is also shown (Framework). We observe that the overhead averaged over all applications and combinations of modules is 53.1% lower than the overhead due to full replication.

Table 2: Overhead for different configurations of modules

	Framework	SELREP	FULLREP
schedule	9.2%	11.9%	36.2%
schedule2	8.1%	11.5%	31.6%
print_tokens	6.9%	20.9%	47.7%
print_tokens2	7.9%	21.8%	46.1%

An average over all the Siemens benchmarks shows that the overhead is 16.5% for SELREP. For SELREP the overhead is due to the execution of duplicate instructions in replicated mode, and due to the switch between replicated and unreplicated modes. The overhead varied from application to application. For example, for *schedule* the overhead for SELREP was 67.1% lower whereas for *print_tokens2* it was 52.7% lower than full replication (FULLREP).

5.3 Error Analysis

In this section we describe the fault-injection analysis of the error coverage provided by Selective Replication. Firstly, the fault model is described and the classification of the outcomes of each fault-injection experiment is presented. Independent experiments are conducted for each benchmark from the Siemens suite.

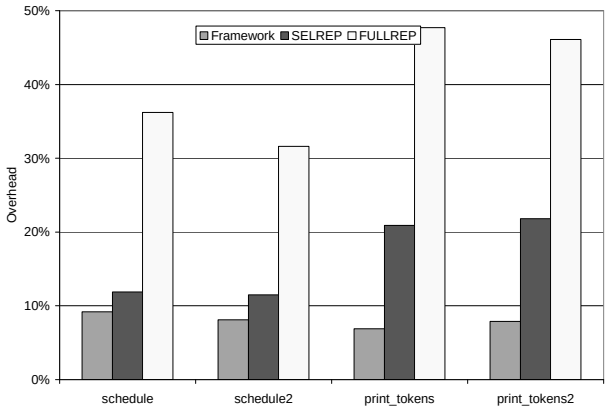

Figure 5: Performance overhead for Selective Replication

5.3.1 Fault model

An important component of the design of a fault-injection experiment to evaluate error coverage is the fault model. It describes the faults that are being targeted by the error-detection mechanisms and against which they have been evaluated. The faults considered in our experiments are as follows:

- *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.
- *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 written to registers.

This fault model also includes some software faults such as *assignment/initialization* (an uninitialized or incorrectly initialized value is used) or *checking* (a check performed on the variable fails, which is equivalent to an incorrect value of a variable being used). The error-detection mechanisms detect the symptoms of errors, irrespective of whether they occur in software or hardware.

5.3.2 Fault Injection Outcomes

The SimpleScalar *sim-outorder* performance simulator simulates the timing information for instructions executing in a pipeline; i.e., it maintains the information of which instructions are present in each stage of the pipeline in any given cycle. The simulator, however, computes the results of executing the instructions in the dispatch stage, when it allocates an entry in the reorder buffer to the instruction. It detects and reports any exceptions that result out of the execution of the instruction at the dispatch stage itself, without waiting until the commit stage. Thus, the

processor simulator does not support precise exceptions. The replication mechanism, however, performs the checks when the instruction has arrived at the commit stage and is ready for commit. Considering this behavior of the processor the fault injection outcomes have been organized into the various categories tabulated in Table 3.

Table 3: Fault Injection Outcomes

Outcome	Description	Error Impact
Replication-Detection	Errors leading to a mismatch between the replicas.	Do not raise an exception, but are detected by the voter in the commit stage.
Exception Raised	Errors that raise a simulator exception in the same instruction (PC) as the injected PC.	Raise an exception in the commit stage of the injected instruction. Architected state would not be corrupted by these errors before the exception.
Retrospectively-Detected	Errors that are not detected by replication, but are injected when the processor is in replicated mode, and raise a simulator error in a different instruction than the one that was injected into.	Can be detected by the replication if the instruction had been allowed to complete. However, the architected state might have been corrupted by then.
System Detection	Errors that are detected by the simulator but occur in a different instruction than the injected instruction.	Detected by the system, but the architected state might have been corrupted by the instruction before the system detects these errors.
Not Manifested	Errors that do not cause simulator errors and hangs, and the output files match.	Do not cause any visible effect on the outcome of the program.
Program Hang	Errors in which the simulator times out and kills the program	Cause the simulator to wait indefinitely for the program to complete
Fail-silent Violation	Errors that do not cause simulator errors or timeouts, but result in the output files, differs from that of the golden run.	System does not detect these errors, but results in an incorrect program outcome. Potentially, most dangerous of the error categories
Benign Error Detection	Errors that are "Not Manifested" in the baseline case but are detected by the detection mechanism	Do not affect application outcome and hence need not be detected

Translating the simulator behavior to that of a real processor, let us assume that the simulator does not raise an exception at the dispatch stage but allows the instruction to proceed to the commit stage.

For the *Exception Raised* category of errors the exception must be reported when the injected instruction is about to commit. If the injected instruction was replicated, the replication mechanism could have detected a mismatch in the results of the instruction at the commit stage before the exception is raised. Therefore, for a real processor system, the subset of *Exception Raised* errors where the injected instruction is replicated can be included into the *Replication Detection* category. The rest of the errors are detected by the system through an exception. The *System Detection* category also contains errors that raise a system exception. However, we continue to maintain the distinction between the exceptions raised at the injected instruction and those that are raised at a later instruction. If the system raises an exception at the injected instruction itself, then architectural state is not updated. We categorize these outcomes as *Exception Raised*. But if the exception was raised at a later instruction, the architectural state would have been updated, possibly with incorrect data. These outcomes are *System Detection*.

For the errors belonging to the *Retrospectively Detected* category the exception is raised in a different instruction, even though the injected instruction is replicated. This is because after the injected instruction completed execution and before it reached the commit stage, the simulator dispatches and executes the instructions following it and raises an exception in one of these succeeding instructions. Again, in a real processor the replication mechanism can detect this category of errors at the commit stage.

5.3.3 Error metrics

The two metrics derived from the fault injection outcomes are the percentage of errors detected by the technique and percentage of false positives, where an error that is benign in the baseline is detected by the technique. For any technique it is desirable to have the most sensitive detection possible and as few false positives as possible, even though these are conflicting goals.

5.3.4 Error coverage for instruction errors

Errors belonging to each type mentioned in Section 5.3.1 are injected. Table 4 presents the detection by selective replication (SELREP), averaged over the applications, when 50 critical variables are used to select the critical instructions to be replicated. The detection of selective replication is compared to the outcomes in the baseline case, when randomized replication (RANDOMREP) is used and when full duplication (FULLREP) is used.

These results show that selective replication of instructions affecting 50 critical variables detects about

87% percent of the faults detected by FULLREP. Yet it incurs 59.1% less overhead and leads to 17.8% fewer benign error detection scenarios as compared to full duplication.

Table 4: Results for instruction error injection with SELREP

Outcome \ Configuration	Baseline	RANDOMREP	SELREP	FULLREP
Activated	489	449	504	500
Not Manifested	41.8%	30.3%	19.0%	18.0%
Replication Detection	0.0%	50.9%	62.5%	71.2%
Exception raised in same instruction	29.2%	24.8%	2.0%	0.0%
Exception raised in different instruction	40.1%	18.4%	4.2%	0.0%
Program Hang	8.8%	4.2%	1.7%	0.1%
Fail Silence Violation	21.9%	7.9%	1.9%	0.6%
Benign Error Detection	0.0%	45.9%	48.5%	59.0%

In Figure 6, the y-axis shows the different outcomes from the injection of instruction errors. The x-axis shows the percentage of errors that fall into each outcome category. From Figure 6 we can see that FULLREP detects about 71.2% of the manifested errors. The rest of the errors raise an exception at the injected instruction itself and hence can be detected by the system and recovered easily. Even though selective replication has a much lower overhead than full duplication, it detects 62.5% of the manifested errors, whereas random replication detects only 50.9% of the errors. When random replication is used, the system detects 17% of the errors in a different instruction, which are difficult to recover from. In the case of selective replication, this contributes to only 4.2% of the errors.

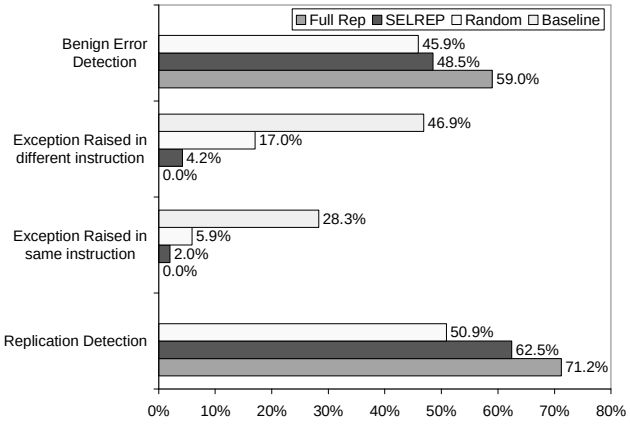

Figure 6: Instruction error injection results

Figure 7 shows the percentage of fail silence violations and program hangs that occur when instruction errors are injected and when different types of replication mechanisms are employed. Full Duplication is able to prevent most of the fail-silence violations and program hangs. Selective Replication is better than randomized replication but worse than full replication in detecting both fail silence violations and program hangs.

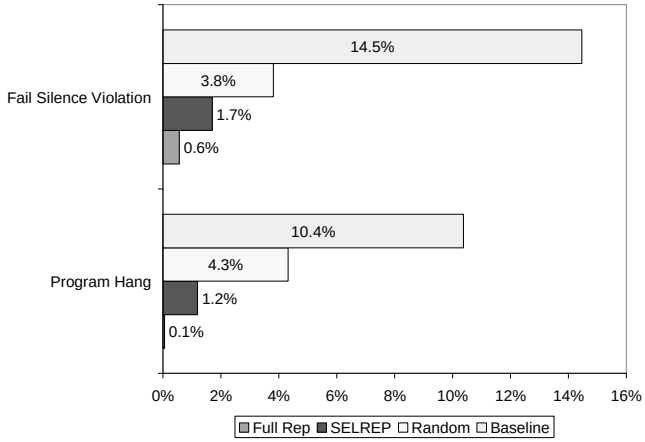

Figure 7: Fail silence violations and program hangs for instruction errors

5.3.5 Error coverage for data errors

Table 5 summarizes the results of injecting data errors (errors into the output of a functional unit when it is generating the result of an instruction). We see that FULLREP detects all the errors. This is because we inject the result of an instruction in only one of the replicas and vote over the result of each replicated instruction. Since all instructions are replicated in FULLREP it can detect all data errors. However, it is important to note that even though only a fraction of the instructions are replicated in selective replication it detects about 97% of all data errors also. From the last row in Table 5 we see that FullRep detects all the data errors that were Not Manifested in the Baseline case, whereas SelRep decreases this detection of errors benign to the application outcome by more than 6%.

Table 5: Fault injection results for data injection

Outcome \ Configuration	Baseline	SELREP	FULLREP
Activated	490	477	477
Not Manifested	44.6%	3.0%	0.0%
Replication Detection	**0.0%**	**97.4%**	**100.0%**
Exception raised in different instruction	72.2%	1.6%	0.0%
Program Hang	2.3%	0.1%	0.0%
Fail Silence Violation	25.5%	0.9%	0.0%
Benign Error Detection	0.0%	93.8%	100.0%

6 Related work

Replicated execution for fault-detection and tolerance has been investigated extensively both at the application and hardware level. At the application-level, instructions in the code segment are duplicated and are expected to execute on the idle hardware in superscalar processors. *Error Detection Using Duplicated Instructions (EDDI)* [2] duplicates of original instructions in the program but with different registers and variables. SWIFT [25] is an application-level duplication mechanism based on EDDI. In *Error Detection by Diverse Data and Duplicated Instructions (ED^{4}I)* [16] two "different" programs with the same functionality but with different data sets, are executed and their outputs are compared. The "different" programs are generated by multiplying all variables and constants in the original program by a diversity factor *k*.

Duplicated at the application level increases the code size of the application in memory. More importantly, it reduces the instruction supply bandwidth from memory to the processor. *EDDI* can possibly be extended to support selective replication by instructing the compiler which portions of the application need to be replicated. *ED^{4}I* has to execute two versions of the same programs and has an effective overhead of more than 100%, since both the applications have to be executed and their results collected and compared.

In the realm of commercial processors Tandem's (now HP) Integrity S2 [16] fault tolerance platform provided triple modular redundancy (TMR) in all hardware components and synchronizing the replicated processors at the interrupt-level. The IBM G5 processor [17] provides duplicate I- and E- units to provide duplicate execution of instructions. The processor is supported by a hierarchical recovery mechanism, from the instruction-level extending upto the system level. To support the duplicate execution, the G5 is restricted to a single-issue processor and incurred 35% hardware overhead.

In experimental research simultaneous multithreading (SMT) [19] and the chip multiprocessor (CMP) architectures have been ideal bases for space and time redundant fault-tolerant designs because of their inherent redundancy. In the AR-SMT architecture fault tolerance is achieved by executing two copies of the same program on an SMT processor [20]. A later work develops similar concepts in the context of CMPs [21]. In simultaneously and redundantly threaded processors (SRT), AR-SMT is further enhanced by checking only instructions whose side effects are visible beyond the boundaries of the processor core. This allows looser coupling between the redundant threads [22]. This modified SMT-based fault-tolerant design is subsequently extended in simultaneously and redundantly threaded processors with recovery (SRTR) to include recovery [5]. Another fault-tolerant processor architecture is proposed in the DIVA design [3][4]. DIVA comprises an aggressive out-of-order superscalar processor along with a simple in-order checker processor. The checker processor verifies the output of the complex out-of-order processor and triggers a recovery action when an inconsistency is found. *Microprocessor-based introspection (MBI)* [23] is a time redundancy technique, to detect transient faults. MBI achieves time redundancy by scheduling the redundant execution of a program during idle cycles in which a long-latency cache miss is being serviced. Even though full duplication at the processor-level has been believed to have little or no performance overhead, [5] and [23] have reported upto 30% overhead. SLICK [26] is a SRT based approach to provide partial replication of an application, compared to this approach we do not rely on a multi-threaded architecture for the replication. Instead, this paper presents modifications to a general superscalar processor to support partial or selective replication of the application.

The basic principle of fault-tolerance employed in all the previous techniques that have been discussed is replication. This is also the focus of this paper. But a major difference is that none of the previous techniques provide a mechanism to dynamically configure the level of replication according to the application's demand. The application also does not have a choice of not replicating part of its code. This requires providing an interface to the application, either at the high-level programming language or at the assembly level, to invoke and configure the replication mechanism at run-time. This is the major contribution of this paper.

7 Conclusion

In this paper we have demonstrated an approach to extract sensitive sections of code that can be selectively replicated to enhance the reliability of the application, instead of replicating the entire application. We have given a detailed design and evaluation of the mechanism to support this selective replication at the processor architecture level. The results show that with about 59% less overhead than full duplication of all instructions we can cover 97% of the data errors and 87% of the instruction errors that were covered by full duplication. An important advantage of the selective replication is that reduces the detection of errors benign to the final outcome of the application by 17.8% as compared to full duplication.

REFERENCES

[1] R. K. Iyer, N. Nakka, Z. T. Kalbarczyk, and S. Mitra, "Recent advances and new avenues in hardware-level reliability support," *IEEE MICRO*, vol. 25, no. 6, pp. 18-29, Nov.-Dec. 2005.

[2] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51(1), pp. 63-75, Mar. 2002.

[3] C. Weaver and T. Austin. "A fault tolerant approach to microprocessordesign," in *Proceedings of the International Conference on Dependable Systems and Networks*, July 2001, pp. 411-420.

[4] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the Thirty-Second International Symposium on Microarchitecture*, November 1999, pp. 196-207.

[5] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient fault recovery using simultaneous multithreading," in *Proceedings of the Twenty-Ninth Annual International Symposium on Computer Architecture*, May 2002, pp. 87-98.

[6] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the Thirty-Fourth Annual International Symposium on Microarchitecture*, Austin, Texas, Dec. 2001, pp. 214-224.

[7] G. Choi, R. Iyer, and V. Carreno, "FOCUS: An experimental environment for validation of fault sensitivity analysis," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1515-1526, Dec. 1992.

[8] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings of Dependable Systems and Networks*, 2004, pp. 61-70.

[9] G. Saggese, A. Vetteth, Z. T. Kalbarcyzk, and R. K. Iyer, "Microprocessor Sensitivity to Failures: Control vs Execution and Combinational vs Sequential Logic," in *Proceedings of Dependable Systems and Networks*, 2005, pp. 760-769.

[10] K. Pattabiraman, Z. T. Kalbarczyk, and R. K. Iyer, "Application-based metrics for strategic placement of detectors," in *Proceedings of Pacific Rim Dependability Conference*, 2005, pp. 8-15.

[11] M. D. Ernst, "Dynamically detecting likely program invariants," Ph.D. dissertation, University of Washington, Department of Computer Science and Engineering, August 2000.

[12] S. Bagchi, S. Narayanaswamy, Z. Kalbarcyzk, and R.K. Iyer, "Design and evaluation of preemptive control signature (PECOS) checking for distributed applications," *Submitted to IEEE Transactions on Computers*, 2002.

[13] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers – Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley Publishers, 1986.

[14] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," University of Wisconsin-Madison, Technical Report CS-1342, June 1997.

[15] M. Hiller, A. Jhumka, and N. Suri, "On the placement of software mechanisms for detection of data errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 135-144.

[16] D. Jewett, "Integrity S2: A fault-tolerant Unix platform," *Digest of Papers Fault-Tolerant Computing: The Twenty-First International Symposium*, Montreal, Canada, pp. 512 - 519, June 25-27, 1991.

[17] T. Slegel, et al. "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19(2), pp. 12–23, 1999.

[18] N. Oh, S. Mitra, and E.J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transactions on Computers*, vol. 51(2), pp. 180-199, Feb. 2002.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip performance," in *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, June 1995, pp. 392-403.

[20] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the Twenty-Ninth International Symposium on Fault-Tolerant Computing Systems*, June 1999, pp. 84-91.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," In *Proceedings of the Thirty-Third International Symposium on Microarchitecture*, December 2000, pp. 269-280.

[22] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the Twenty-Seventh International Symposium on Computer Architecture*, June 2000, pp. 25-36.

[23] M. A. Qureshi, O. Mutlu, and Y. N. Patt, "Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors," In

Proceedings of International Conference on Dependable Systems and Networks, June 2005, pp. 434-443.

[24] N. Nakka. "Reliability and Security Engine: A Processor-level framework for Application-Aware detection and recovery," PhD dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urban-Champaign, USA, 2006.

[25] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the3rd International Symposium on Code Generation and Optimization*, March 2005.

[26] A. Parashar, A. Sivasubramaniam, S. Gurumurthi. "SlicK: slice-based locality exploitation for efficient redundant multithreading," in Proceedings of the 12th Intl., conference on ASPLOS, 2006.