DRIFT: Decoupled compileR-based Instruction-level Fault-Tolerance [⋆]

Konstantina Mitropoulou [†], Vasileios Porpodas[†] and Marcelo Cintra[†⋆]

School of Informatics, University of Edinburgh [†]
Intel Labs Braunschweig[⋆]
{K.Mitropoulou@sms., v.porpodas@, mc@staffmail.}ed.ac.uk

Abstract. Compiler-based error detection methodologies replicate the instructions of the program and insert checks wherever it is needed. The checks evaluate code correctness and decide whether or not an error has occurred. The replicated instructions and the checks cause a large slowdown. In this work, we focus on reducing the error detection overhead and improving the system's performance without degrading fault-coverage. DRIFT achieves this by *decoupling* the execution of the code (original and replicated) from the checks.
The checks are compare and jump instructions. The latter ones sequentialize the code and prohibit the compiler from performing aggressive instruction scheduling optimizations. We call this phenomenon *basic-block fragmentation*. DRIFT reduces the impact of basic-block fragmentation by breaking the synchronized execute-check-confirm-execute cycle. In this way, DRIFT generates a scheduler-friendly code with more ILP. As a result, it reduces the performance overhead down to 1.29× (on average) and outperforms the state-of-the-art by up to 29.7% retaining the same fault-coverage. The evaluation was done on an Itanium2 by running MediabenchII and SPEC2000 benchmark suites.

Keywords: compiler error detection, fault tolerance

1 Introduction

The current techniques to improve performance and to reduce energy consumption have made transistors more vulnerable to errors [6][24][29]. Soft Error Rate (SER) increases as we move to small transistor technologies. In addition, techniques like voltage scaling require transistors to operate at their voltage limit. This increases SER further. An important class of hardware errors is transient errors (a.k.a. soft errors) which occur only once and do not persist [28]. Although transient errors are temporal phenomena, they can alter the program's execution. For instance, in 2000, Sun Microsystems received several complaints from customers such as America On-line, eBay, and Los Alamos Labs, who experienced system failures because of transient errors [18].

Hardware redundancy based error detection techniques are used in high-availability systems and mission critical environments. Typical examples are IBM's G4 and G5 processors [26] and HP NonStop series processors [4]. Not all systems can afford the cost of the extra hardware and design complexity of

[⋆] This work was supported in part by the EC under grant ERA 249059 (FP7).

hardware-based error detection. Compiler-based error detection might be preferable instead. There are several reasons: 1. It is more flexible and cheaper than the hardware design and it can be applied on-the-fly on any system. 2. It operates at a higher abstraction level restricting the error detection only to errors that might affect application's output. 3. It gives the designer the flexibility to choose the program region that he wants to protect. Its main drawback is that code duplication has negative impact on performance.

High fault-coverage compiler-based error detection (ED) methodologies face the challenge of effectively managing the error detection overhead without sacrificing reliability. There are two approaches to this. **Synchronized** techniques require that the original and redundant code execute in sync such that the execution is checked in strict intervals. In this way, the strict synchronization guarantees fail/stop behavior, but it has negative impact on the code's performance. On the other hand, **decoupled** approaches remove the strict synchronization between the original and the redundant code, and they let them slip against one another, while performing the checks slightly later, when convenient. Thus, the program runs faster. However, the system looses its fail-stop capability since the synchronization points are removed.

Compiler-based ED techniques increase the code size since they generate redundant and checking code. This extra code can be executed either on the same processor as the original code (**single-core** techniques) or on a separate core (**dual-core** techniques). Each scheme is suited for different use scenarios. On one hand, if there are spare cores and no energy restrictions (all the cores are turned on), then the dual-core technique is the best option. On the other hand, if there are no free cores, or the application is one that benefits from using multiple cores, then wasting multiple cores for running the redundant code is not wise. Under these circumstances, it might be preferable to apply the single-core ED scheme on each thread of the multi-threaded application or each program of a multi-programmed workload. DRIFT is an improved single-core technique.

Our work is based on the observation that the frequent checking of the synchronized scheme becomes a performance bottleneck. This is a phenomenon we refer to as **basic-block fragmentation**. The checks break the code into very small basic-blocks with two exiting control edges (Figure 1.1.b). The resulting complex control flow acts as a barrier for aggressive compiler optimizations at the instruction scheduling level, even for the most aggressive schedulers. For example, in Figure 1.1, the original basic-block BB1 (Figure 1.1.a) splits into three basic-blocks. The scheduler cannot easily move the instructions among basic-blocks to improve ILP because it strictly must respect the program semantics. This is an important restriction that prohibits the compiler from generating high performance code for synchronized single-core ED. **DRIFT** introduces a novel decoupled single-core technique that avoids the basic-block fragmentation and improves the performance considerably by relaxing the synchronization between original, replicated code and checks. It achieves this by clustering the checks (Figure 1.1.c) so as to keep the basic-blocks big. In this way, the code is not fragmented into many basic-blocks and can be scheduled more efficiently.

We note that, strictly speaking, our code generation scheme does modify the code semantics. This, however, takes advantage of knowledge of ED semantics (which are not available to a standard compiler) and does affect the semantics of the original (non-ED) program. Therefore, the aggressive code motion that we perform in DRIFT, could not have been done automatically by any compiler optimization since the compiler is restricted to always preserving the program semantics.

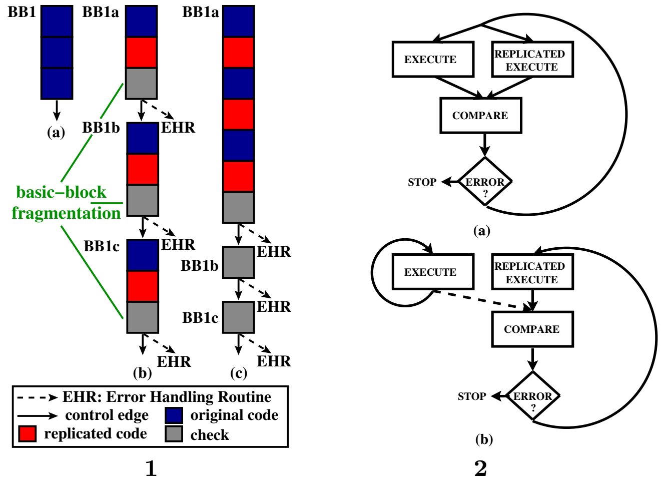


Fig. 1: 1.Control-flow graph for (a) code without ED, (b) synchronized ED (SWIFT) and (c) decoupled ED code (DRIFT). 2.(a) Synchronized ED and (b) Decoupled ED.

Our contributions are:

- This work is the first to point out a major performance bottleneck in synchronized ED caused by basic-block fragmentation.
- DRIFT overcomes the basic-block fragmentation bottleneck by being the first decoupled single-core ED scheme.
- DRIFT outperforms the state-of-the-art by up to 29.7% reducing the performance overhead down to 1.29× while retaining high fault-coverage.

The rest of the paper is organized as follows: Section 2 presents basic-block fragmentation problem and the proposed solution. Section 3 describes DRIFT algorithm. Section 4 shows the experimental set-up. Section 5 discusses performance and fault-coverage results. Section 6 overviews the related work. Section 7 concludes this paper.

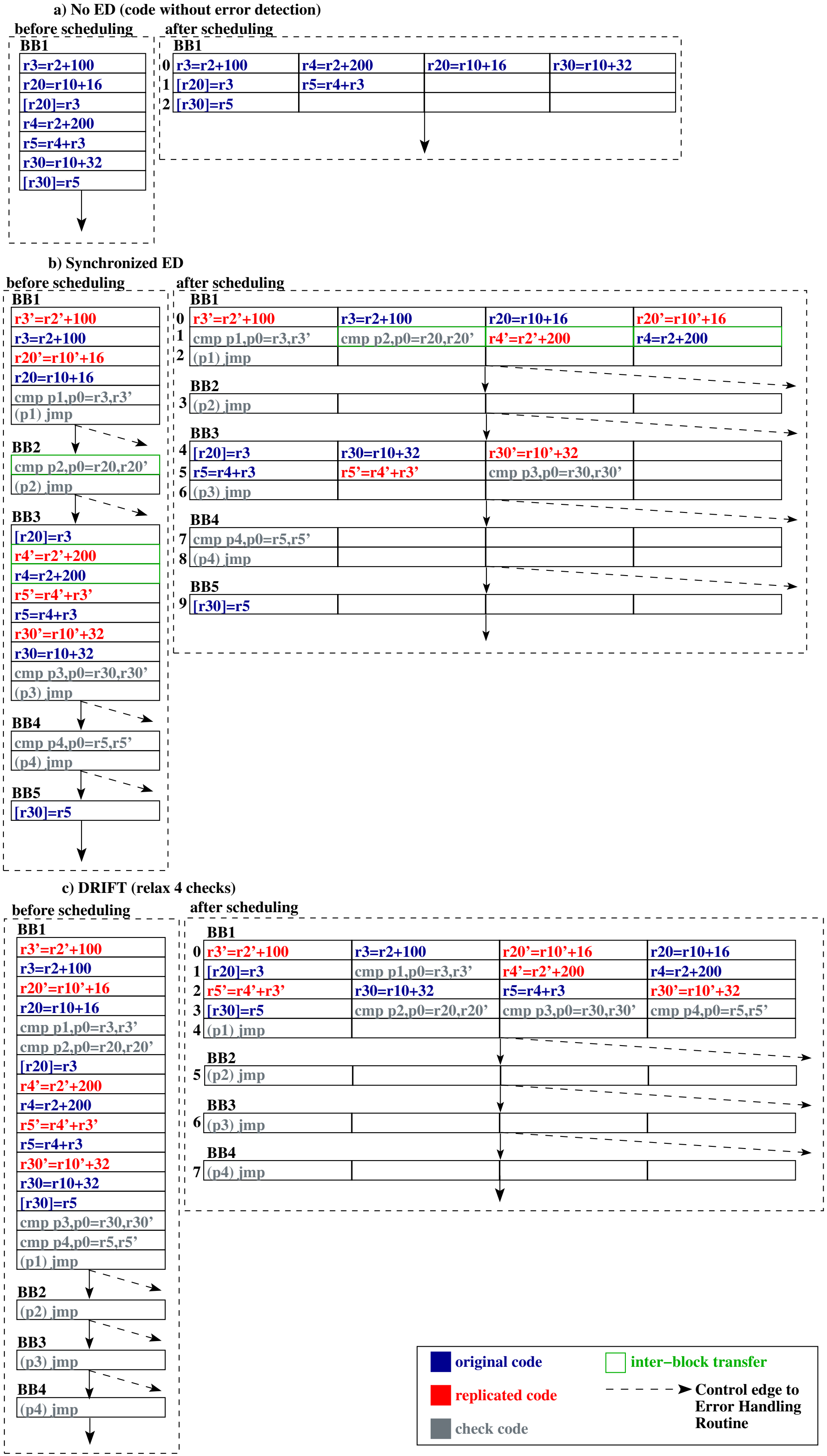
2 Motivation

Synchronized VS Decoupled: In compiler-based error detection, decoupling was first used in DAFT[32] so as to remove the overhead of synchronizing between the main and the checker thread. In that case the main and checker threads are decoupled and allowed to slip between each other. In synchronized single-core error detection, checks are synchronization points where the code is checked for errors and a control point is inserted in the code. In Figure 1.2.a, it is shown that the execution of the program is interrupted by the checks, which are in the critical path of the program. Therefore, the need to synchronize very often is a significant slowdown factor for compiler-based error detection. The solution is to remove these synchronization points by decoupling the execution of the code (original, replicated) from the checks. In Figure 1.2.b, we see that the program does synchronize since the checks can be executed some time later. This boosts the performance of the program and reduces the error detection overhead. Such performance improvement may come at the expense of reduced fault coverage. However, as shown previously (e.g., [32]), the impact on fault-coverage is not serious. This is further discussed in the Decoupled Single-Core (DRIFT) Section.

Synchronized Single-Core limitations: In the Synchronized ED (Figure 1.1.b), all original, replicated and checking code is placed on the same thread. A check is placed right before a non-replicated instruction. Every check compares the original and the replicated code using a compare (CMP) instruction. If the check succeeds, then the code continues executing (no jump), otherwise the control jumps (JMP) to the appropriate error handling routine.

The performance bottleneck of this scheme due to such synchronization, shows up as what we call *basic-block fragmentation*. This problem has two main factors: *1.The complicated Control Flow:* The frequent checks (CMP + JMP) break the original code into a sequence of small basic-blocks with two outgoing edges each. For example BB1 in Figure 2.a gets split by ED into five basic-blocks (Figure 2.b). *2.Instruction scheduling:* The complex control flow due to the checks acts as a scheduling barrier for the instruction scheduling optimization (e.g., trace scheduling). Even with a speculative scheduler that schedules regions of multiple basic-blocks, the control edges (due to the checks) limit the scheduler's ability to hoist instructions and extract adequate amounts of ILP. Any state-of-the-art region-based instruction scheduler has some *limitations* in hoisting instructions across basic-blocks: *1.*It cannot hoist instructions with side-effects over branches since this can break the program semantics. This restricts the hoisting of system calls, and store instructions [11, 14]. *2.*If there is no hardware support for deferring exceptions then dangerous instructions such as loads and divisions cannot be hoisted either [15]. As a result, the scheduler generates poorly performing schedules, with low ILP.

Decoupled Single-Core (DRIFT): In this paper we propose DRIFT, an ED scheme that addresses the shortcomings of the Single-Core Synchronized scheme, as described earlier. DRIFT is based on three ideas: *1.Optimized Control Flow:* Modifying the control flow of the application can enhance the ability of the instruction scheduler to optimize the code. Since instruction schedulers are not as effective across basic-blocks as within basic-blocks, larger basic-blocks are better. This can be done by decoupling the execution of checks and by executing them later together as a group. By contrasting Figure 2b versus Figure 2c, we observe that DRIFT generates a much more instruction-scheduler friendly code than the Synchronized scheme. *2.It is acceptable to break the semantics of the combined original and replicated code, as long as the semantics of the original code are respected.* This unawareness of normal compilers to the semantics of ED code is the main reason why the compiler cannot automatically generate decoupled code (like the one DRIFT generates) out of the synchronized code. Therefore the code of Figure 2c cannot have been generated by any compiler optimization. Breaking the semantics in a controlled way is required for modifying the code in such an aggressive way. *3.DRIFT's decoupled semantics have no effect on fault-coverage.* As shown in [32], modifying the semantics of the application with ED support, such that the checks are decoupled from the execution, has a minimal impact on the effectiveness of error detection. This is because in the usual case, the increased delay between the error and its detection is not great



a) No ED (code without error detection)

before scheduling

BB1

r3=r2+100
r20=r10+16
[r20]=r3
r4=r2+200
r5=r4+r3
r30=r10+32
[r30]=r5

after scheduling

BB1

0	r3=r2+100	r4=r2+200	r20=r10+16	r30=r10+32
1	[r20]=r3	r5=r4+r3		
2	[r30]=r5			

b) Synchronized ED

before scheduling

BB1

r3’=r2’+100
r3=r2+100
r20’=r10’+16
r20=r10+16
cmp p1,p0=r3,r3’
(p1) jmp

BB2

cmp p2,p0=r20,r20’
(p2) jmp

BB3

[r20]=r3
r4’=r2’+200
r4=r2+200
r5’=r4’+r3’
r5=r4+r3
r30’=r10’+32
r30=r10+32
cmp p3,p0=r30,r30’
(p3) jmp

BB4

cmp p4,p0=r5,r5’
(p4) jmp

BB5

[r30]=r5

after scheduling

BB1

0	r3’=r2’+100	r3=r2+100	r20=r10+16	r20’=r10’+16
1	cmp p1,p0=r3,r3’	cmp p2,p0=r20,r20’	r4’=r2’+200	r4=r2+200
2	(p1) jmp			

BB2

3	(p2) jmp			

BB3

4	[r20]=r3	r30=r10+32	r30’=r10’+32	
5	r5=r4+r3	r5’=r4’+r3’	cmp p3,p0=r30,r30’	
6	(p3) jmp			

BB4

7	cmp p4,p0=r5,r5’			
8	(p4) jmp			

BB5

9	[r30]=r5			

c) DRIFT (relax 4 checks)

before scheduling

BB1

r3’=r2’+100
r3=r2+100
r20’=r10’+16
r20=r10+16
cmp p1,p0=r3,r3’
cmp p2,p0=r20,r20’
[r20]=r3
r4’=r2’+200
r4=r2+200
r5’=r4’+r3’
r5=r4+r3
r30’=r10’+32
r30=r10+32
[r30]=r5
cmp p3,p0=r30,r30’
cmp p4,p0=r5,r5’
(p1) jmp

BB2

(p2) jmp

BB3

(p3) jmp

BB4

(p4) jmp

after scheduling

BB1

0	r3’=r2’+100	r3=r2+100	r20’=r10’+16	r20=r10+16
1	[r20]=r3	cmp p1,p0=r3,r3’	r4’=r2’+200	r4=r2+200
2	r5’=r4’+r3’	r30=r10+32	r5=r4+r3	r30’=r10’+32
3	[r30]=r5	cmp p2,p0=r20,r20’	cmp p3,p0=r30,r30’	cmp p4,p0=r5,r5’
4	(p1) jmp			

BB2

5	(p2) jmp			

BB3

6	(p3) jmp			

BB4

7	(p4) jmp			

Fig. 2: The code before and after instruction scheduling for (a) code without ED, (b) synchronized ED and (c) DRIFT where four checks are executed together.

enough to let the error propagate to the output. Moreover, it has been shown in [7][13][31] that a significant number of errors such as ISA-defined exceptions can be detected by the operating system. This is a fundamental feature of DRIFT, which guarantees its high fault-coverage despite the modified semantics that allow for better performance.

DRIFT motivating example: In Figure 2, the example shows the code for 3 cases: *1.* No error detection, *2.* Synchronized ED and *3.* DRIFT which decouples 4 checks. Each sub-figure shows the code before instruction scheduling (left) and the scheduling table (right) of a hypothetical 4-issue machine. All ED, schemes Figure 2.b-c, contain the same number of checks and replicated instructions (red). This is because all schemes have the same sphere of replication (see Section 3). In this work, our baseline is SWIFT [22] which is the state-of-the-art single-core error detection. In SWIFT, checks are added before store instructions. For example the store instruction “[r20]=r3” has its inputs checked. The check “cmp p1,p0 = r3,r3”' makes sure that the instructions “r3=r2+100” and “r3'=r2'+100” produce the same result.

Basic-Block Fragmentation: Checks split the code into numerous basic-blocks. For example the original code of Figure 2.a is a single basic-block, but the ED code of Figure 2.b spans over 5 basic-blocks (BB1-BB5). Therefore, checks act as fragmentation points for the Control Flow Graph (CFG).

The difference between the Synchronized scheme (Figure 2.b) and the DRIFT scheme (Figure 2.c) is the amount of fragmentation of the basic-blocks. The Synchronized case is the most fragmented one, as checks are regularly injected into the code (see Figure 2.b left). On the other hand, DRIFT groups together multiple checks. In the example of Figure 2.c, it groups 4 checks together. We refer to this grouping of checks as “decoupling” and the amount of checks being decoupled as “decouple factor”. Increasing the check relaxation decreases the fragmentation of basic-blocks (see Figure 2.c left).

Performance and Schedule: To understand the impact of decoupling on performance, we have to look into the instruction schedule tables (on the right side of each sub-figure). The schedule is obtained after an inter-block instruction scheduler has scheduled across the basic-blocks of the ED code (left). Inter-block code hoisting is marked with green. The Synchronized scheme is fragmented as checks introduce edges into the control flow. These edges prohibit aggressive code hoisting in several cases. For example, “[r20]=r3” of BB3 cannot be hoisted into BB2 or BB1 as it modifies unknown memory. For the same reason, “r[30]=r5” of BB5 cannot be hoisted as well.

Removing these control flow restrictions improves the schedule considerably, as instructions can be hoisted and parallelized easily. For example in Figure 2.c, all instructions are within a single basic-block (BB1), which makes it straightforward for any scheduler to parallelize.

3 DRIFT

Sphere of Replication: Similar to other state-of-the-art compiler-based ED techniques ([22],[30],[32]) DRIFT assumes that the memory is protected by its own mechanisms like Error Correcting Code (ECC), parity checking or other mechanisms. Therefore the data fetched from the memory is considered to be correct. Thus the Sphere of Replication (SoR) in DRIFT is limited to within the processor only.

The instructions that are not replicated are: *1.*Control Flow instructions (e.g., branches, function calls). *2.*Store instructions.

The code of the linked binary libraries is not protected. This can be changed by recompiling them with DRIFT.

The non-replicated instructions are synchronization points since the checks are inserted before them.

Decouple Factor: As explained in Section 2, DRIFT decouples the checks off the critical path of the execution by grouping them. Each group of checks contains up to N number of checks. We refer to this as decoupling N checks or setting the *decouple factor* to N. Therefore the decouple factor is a knob that controls the number of checks that are executed later together in a group. For example, if the decouple factor is two, then the checks will execute in pairs. For small values of the decouple factor, the program has similar (though slightly better) behavior to the Synchronized ED and suffers from basic-block fragmentation. As the decouple factor increases, more checks are clustered together giving the scheduler the freedom to schedule the instructions more efficiently.

Increasing the decouple factor has two side-effects: 1. We slightly increase the risk of allowing erroneous data to propagate to memory and corrupt the output of the program. 2. We keep more values in predicate registers which increases the predicate register pressure. This may cause performance degradation if it results in register spilling. Moreover, for big values of decouple-factor, many checks are executed together. This means that there might not be enough units to deal with this workload. Therefore, there is a trade-off between the number of checks that are decoupled, the fault-coverage and the hardware capacity. We explore the effect of the decoupled factor on both performance and reliability in the results Section.

DRIFT algorithm is listed in Algorithm 1 and it operates in four steps:

1.Code Replication: The algorithm checks if an instruction can be replicated (Algorithm 1.a line 11). If this is true, then an exact duplicate of the original instruction (Algorithm 1.a line 13) is emitted just before the original one. The original instruction and its replica are inserted into a table (Algorithm 1.a line 14). This table is used later in the algorithm to recall the replicated instruction that corresponds to any original instruction.

2.Code Isolation: This step isolates the replicated code from the original code (Algorithm 1.a line 17). The isolation makes sure that the replicated code does not write on any of the original code's registers. Register isolation does not let the replicated code affect the original code's execution in any way. This is done by register renaming the replicated instructions. In short, the algorithm iterates over all original instructions in the program (Algorithm 1.a lines 18,19) and for each of them it retrieves the corresponding replicate instruction from the table (see step 1) (Algorithm 1.a line 21) and renames all registers written by the replicated instructions along with each of their uses (Algorithm 1.a line 22). All renamed registers are filled into a table which is used in step 3.

3.Emit checks: Next, the algorithm finds all the non-replicated instructions. For each non-replicated instruction (Algorithm 1.b line 4), the algorithm finds the registers that the non-replicated instruction reads. For each one of these registers (Algorithm 1.b line 5), it emits one compare instruction right before the non-replicated instruction. The compare instruction compares the original register against the corresponding renamed one (it gets it by accessing the data-structure of step 2). The synchronized ED technique emits a jump instruction immediately after the compare instruction, it updates the control-flow and this is the final step of the algorithm. On the other hand, DRIFT collects all the compare instructions of a basic-block into the vector (CMP_VEC) which is used in step 4 to perform the grouping.

4.Decouple Checks: This function (Algorithm 1.b line 10) emit as many jump instructions as the value of decouple factor. In more details, we push the instructions of CMP_VEC into vector GROUP (line 12), until we either reach the maximum group capacity (= DECOUPLE_FACTOR) (line 13) or we reach the end of the basic-block (line 14). Once one of the above occurs, a jump is emitted for each instruction in the group (line 15). For example, if the decouple factor is two and the length of CMP_VEC is six, then the conditional jumps will be emitted in three pairs: two jump instructions are placed after the second, the forth and the sixth compare instruction.

Algorithmorithm 1.a

```
1  relaxed_main (DECOUPLE_FACTOR)
2  {for each BB
3    replicate_insns (BB)
4    register_rename (BB)
5    CMP_VEC = emit_compare_insns (BB)
6    emit_jump_insns (CMP_VEC,
          ↪DECOUPLE_FACTOR, BB)
7  }
8  /*Emit replicated instructions*/
9  replicate_insns (BB)
10 {for INSN in BB instructions
11   skip if INSN i) control-flow
12                ii) memory
13   emit an exact duplicate of INSN
          ↪just before it
14   add the original and the duplicate
          ↪ into the data structure
15 }
16 /*Code isolation.*/
17 register_rename ()
18 {for INSN in BB instructions
19   skip duplicates
20   INSN_ORIG = INSN
21   INSN_DUP = get_duplicate_of (
          ↪INSN_ORIG)
22   rename_writes_and_uses (INSN_ORIG,
          ↪ INSN_DUP)
23 }
```

Algorithmorithm 1.b

```
1  /* Inject the CMP instructions. */
2  emit_compare_insns (BB)
3  {for INSN in instructions:
4     skip all but the non-replicated
            ↪instructions.
5     for each REG read by INSN:
6      Get REG_RENAMED(the renamed REG
             ↪from the data structure).
7      Emit CHECK_INSN before INSN
             ↪comparing REG with
             ↪RENAMED_REG.
8  }
9  /*Decouple checks.*/
10 emit_jump_insns (CMP_VEC,
         ↪DECOUPLE_FACTOR, BB)
11 {for CMP_INSN in CMP_VEC
12   push CMP_INSN into GROUP
13   if(GROUP has DECOUPLE_FACTOR
           ↪members
14      OR end of BB reached)
15      Emit JMP_INSN.
16      Update Control Flow Graph.
17 }
```

Processor: Itanium2		Cache (same as Itanium[17])				
Issue width	6	Levels	L1	L2	L3	Main
Instruction Latencies	Same as Itanium2 [17]	Size	16KB	256KB	3MB	∞
Register File	128GP, 128FL, 64PR	Block size	64B	128B	128B	-
Branch Prediction	Perfect	Associativity	4-Way	8-way	12-way	-
		Latency(cycles)	1	5	12	150

Table 1: SKI IA64 configuration.

4 Experimental Setup

We implemented our error detection scheme in a compiler pass in GCC-4.5.0 [1]. The DRIFT pass was placed just before the first instruction scheduling pass.

We evaluated our compiler-based error detection scheme using 9 benchmarks from the Mediabench II video [8] and the SPEC CINT2000 [10] benchmarks. These are the benchmarks that we managed to compile with our heavily modified compiler.

All benchmarks were compiled with -O2 optimizations enabled. To prevent optimizations such as Common Sub-expression Elimination (CSE) and Dead Code Elimination (DCE) from removing the replicated code, we disabled them at the *late* back-end stages of compilation, only for the ED schemes (they are enabled in NOED). This is common-practice in compiler-based error detection schemes (e.g., SWIFT [22]). The performance impact of these disabled phases is negligible (1.5% in the worst case and 0.3% on average).

The performance evaluation was done on a DELL PowerEdge 3250 server with 2x1.4GHz Intel Itanium 2 processors. For the fault coverage evaluation, we used a modified SKI IA-64 simulator [2] (Table 1). The simulator is a cycle-accurate Itanium 2 simulator, modified to allow fault injection.

5 Results and Analysis

We evaluated our scheme by measuring: *1.NOED* which is the code with no error detection, *2.SWIFT* which is the state-of-the-art synchronized single-core error detection methodology [22]. For simplicity, SWIFT is usually implemented with branch checking instead of control-flow checking [5][7]. These techniques have the same overhead. The only difference is that control-flow checking verifies the execution of a jump instruction. It should be noticed that data checking is orthogonal to control-flow checking. This means that control-flow checking can be plugged in the proposed technique as well without any performance degradation. *3.DRIFT* was implemented with various decouple factors (DEC-2, DEC-4, DEC-8, DEC-16, DEC-INF). For example, DEC-4 implies a decouple factor of four. DEC-INF implies an infinite decouple factor which suggests that all checks are placed at the end of the basic-block. A decouple factor of 1 is not measured because it is equivalent to SWIFT.

DRIFT can be applied to multi-threaded applications to protect each of the running threads. In some cases, scalable multi-threaded applications can benefit more from single-core ED than dual-core ED. Because, in the latter case, half of the cores will be used for ED only, hindering the scalability of the application. A detailed comparison against a dual-core scheme is beyond the scope of this paper.

The results are shown in Figure 3 and Figure 4. Each row shows the results of each benchmark. The first column shows the normalized cycle count of all schemes. The cycles are normalized to NOED. The second column presents the percentage of basic-blocks that have a given number of checks. For example, in cjpeg, over 30% of the basic-blocks have 2 checks (checks2). This measurement is based on run-time information (we take into account the number of times each basic-block is executed at run-time). The number of checks usually implies the basic-block size. The last column shows the fault-coverage for all the configurations.

5.1 Performance Evaluation

The results of the first column in Figure 3 and Figure 4 validate our assumption that basic-block fragmentation is a significant slow-down factor of the synchro-

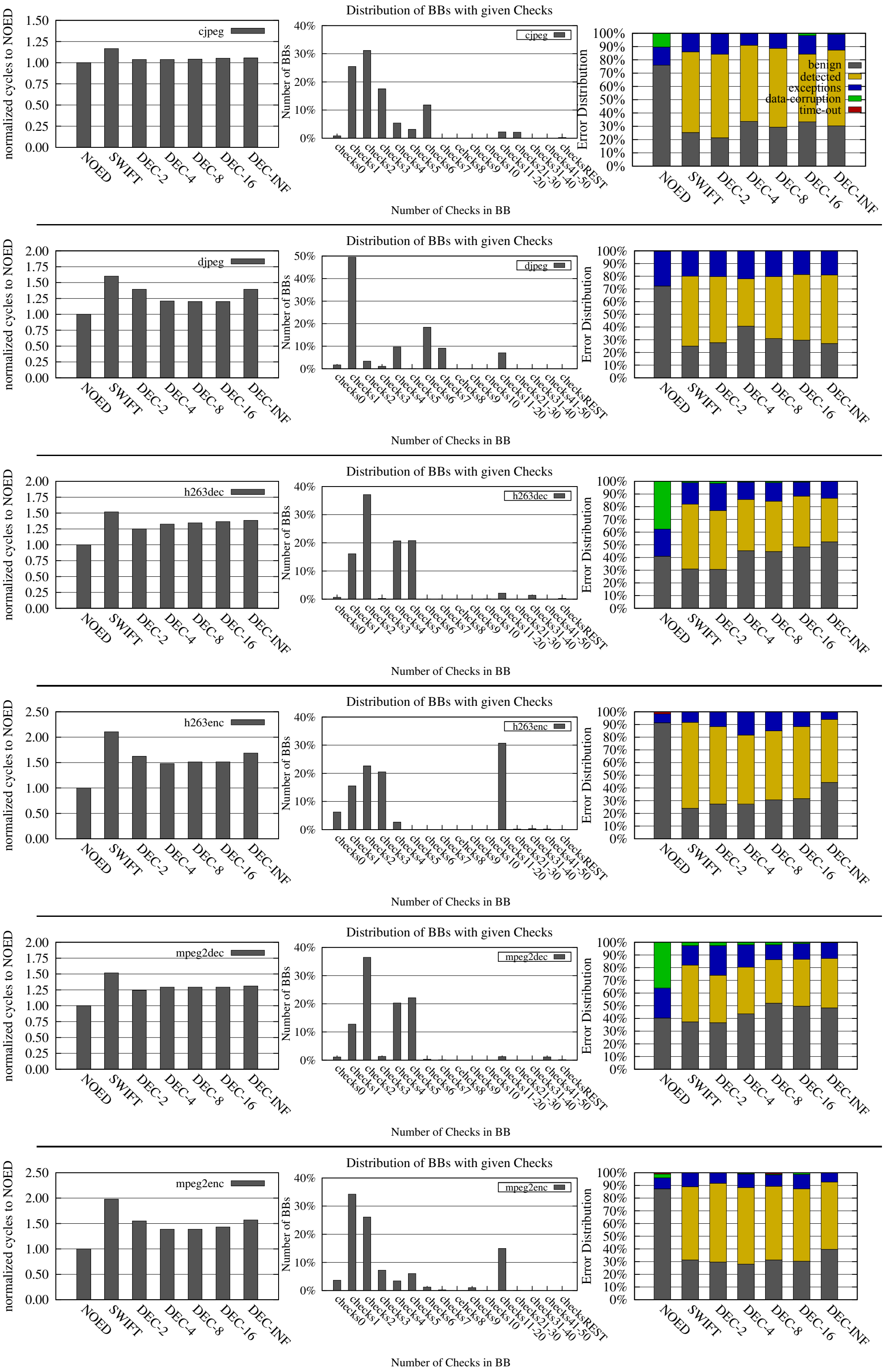


Fig. 3: Results Part 1: The first column shows the performance improvement of DRIFT over SWIFT and NOED, the second one presents the percentage of basic-blocks that have a given number of checks and the third one shows the fault-coverage.

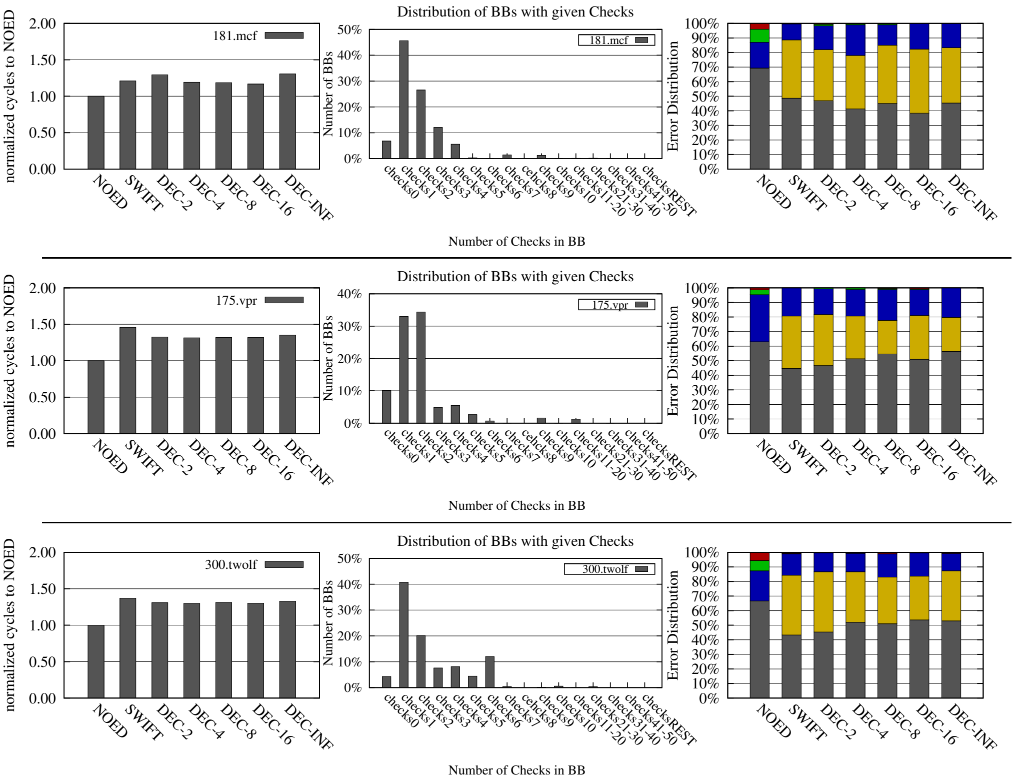


Fig. 4: Results Part 2: Same as Part 1

nized single-core ED scheme (SWIFT). Both techniques were scheduled with the same state-of-the-art GCC region-based speculative scheduler. In the case of SWIFT, it is shown that the compiler cannot produce efficient code since the complicated control-flow acts as a barrier to code motion optimizations. On the other hand, DRIFT creates a scheduler-friendly code. As a result, the performance improvement of DRIFT over SWIFT is up to 29.7% (h263enc, DEC-4) and DRIFT manages to decrease its overhead over NOED down to 1.29×.

DRIFT's performance varies across benchmarks and it is largely affected by the check distribution. Benchmarks like cjpeg, h263dec, mpeg2dec, 175.vpr and 300.twolf have small number of checks per basic-block. Therefore, a decouple factor of 2 is enough to improve their performance. On the other hand, a larger decouple factor benefits the applications that contain many checks per basic-block (e.g., djpeg, h263enc and mpeg2enc).

The performance of some benchmarks, however, degrades as the decouple factor reaches very high values (close to DEC-INF). This is the case for djpeg, h263enc and mpeg2enc. These benchmarks have many basic-blocks with a high number of checks (as shown in the second column). A high value of the decouple factor in these cases can lead to high predicate register pressure. In addition, in the end of each basic-block, we have a tree of compare instructions that slows down the code. That's why DEC-4 performs best for h263enc and mpeg2enc (29.7% and 28% respectively) and DEC-INF is much worse.

Table 2 shows the decouple factor for which DRIFT achieves the best speedup over SWIFT. From the above discussion, we can see that the best decouple factor is a trade-off between basic-block fragmentation and register pressure. The results show that DEC-4 is a good compromise between the two; DEC-4 is big enough to reduce the impact of basic-block fragmentation and small enough to avoid register pressure.

Benchmark	Performance gain over SWIFT	Slowdown over NOED	Decouple Factor	Benchmark	Performance gain over SWIFT	Slowdown over NOED	Decouple Factor
cjpeg	11.1%	x1.04	2,4	mpeg2enc	28%	x1.39	4,8
djpeg	25%	x1.2	8,16	181.mcf	2%	x1.18	8
h263dec	17.7%	x1.25	2	175.vpr	10.5%	x1.31	4
h263enc	29.7%	x1.48	4	300.twolf	5.1%	x1.37	4
mpeg2dec	18.2%	x1.24	2				

Table 2: DRIFT's best performance compared to SWIFT and NOED.

Figure 5 shows that the binary size of SWIFT is about 2.5× greater than NOED. This is expected due to the additional ED code injected into the code stream. DRIFT generates slightly smaller binaries (2.3× greater than NOED), which is further evidence that DRIFT improves the resulting schedule, because the instructions are packed into fewer instruction bundles. As the decouple factor increases the binary size is almost the same. Increasing the decouple factor in benchmarks with small number of checks per basic-block does not change the code any further. In benchmarks (e.g., djpeg, h263enc and mpeg2enc) with large number of checks per basic-block, the ILP might increase as the decouple factor increases, leading to more compact code, but the register spilling adds extra code which counterbalances the code reduction.

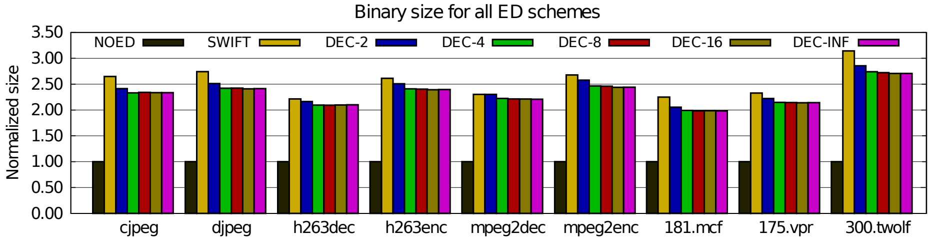


Fig. 5: Binary code size for all benchmarks, normalized to NOED.

5.2 Fault Coverage Evaluation

The fault coverage results presented in this paper are generated using SKI IA-64 simulator [2]. The simulator was modified to inject errors at the output registers of instructions, which is common practice in the literature ([5],[7],[22],[30],[32]).

The fault coverage results are produced with Monte Carlo simulations. The procedure starts with each original binary being profiled in order to count the number of dynamic instructions. The fault injection is done as follows: a dynamic instruction is randomly selected and one of its outputs is randomly picked for injection. Then, a random bit of the register output is flipped. Errors are injected into general purpose and predicate registers. This process is repeated 300 times for each benchmark and each configuration.

For our evaluation, we assume a Single Event Upset (SEU) fault model (double-events are extremely rare [22]). This means that original binaries are injected with one error per run. The binaries that support error detection are much larger (2.3x larger on average than the original (Figure 5)). A fair comparison between the original code and the error detection code requires keeping the error rate fixed [22]. Thus, the error detection codes are injected with one error per the number of dynamic instructions of the original binary. It has to be mentioned that, with this methodology, we do inject errors in the system libraries which are out of DRIFT's SoR.

The output of each Monte Carlo trial is classified into one of the following five categories. *1.Benign Errors* (aka masked errors) are the errors that do not affect program's output and they produce the same output and exit code as the good execution. *2.*The errors that DRIFT algorithm successfully detects are classified as *Detected*. *3.Exceptions* are caught by our custom exception handler and are considered as detected (e.g., as in [32]). *4.Data Corrupt Errors*, which are the errors that cause wrong outputs without being detected. *5.*Finally, some errors result in infinite execution. Those errors are detected by our simulator and we name them *Time out Errors*.

The third column of Figure 3 and Figure 4 shows that DRIFT and SWIFT are almost identical in fault-coverage. In a few cases (h263enc and 181.mcf), some of the detected errors in SWIFT are transformed into exceptions in DRIFT. As we explained in Section 3, both SWIFT's and DRIFT's Sphere of Replication does not include store instructions. Therefore, store instructions are not replicated. In SWIFT, a check is inserted before every non-replicated instruction in order to prohibit corrupted data to propagate to memory. DRIFT delays the execution of some of the checks. Thus, some stores might be executed before verification takes place, leading to exceptions raised by the system. These exceptions are detected by our exception handler (as done in DAFT [32]). As in all high fault-coverage techniques, Data-corruption and Time-out errors are very rare. Therefore, DRIFT has practically the same fault-coverage as SWIFT even for high values of the decouple factor.

In the performance evaluation (Section 5.1), we showed that a decouple factor of 4 always improves system performance. The fault coverage results show that it has very good fault-coverage as well.

Finally, we observe that the *computational nature of the benchmark* plays an important role on fault coverage. For example, mpeg2enc, cjpeg and h263enc, are encoding benchmarks which means that lots of data get compressed. This may involve the process of sub-sampling, which by definition ignores the value of parts of the input. If an error occurs on data that gets compressed, then it may not propagate at all and it will not appear in the output of the program. For this reason, NOED has almost 90% benign errors. In this type of applications, decoupling is less risky.

6 Related Work

Code redundancy can take various forms: instruction, thread and process redundancy. EDDI [20] was the first to introduce **compiler-base instruction-level** redundancy. SWIFT [22] significantly improves upon it by reducing the memory overhead. SRMT [30] and DAFT [32] reduce overhead further by allocating the replicated code and the checks to a second core.

Hardware Thread-level redundancy was introduced by AR-SMT [23]. This work proposed the idea of redundant multi-threading (RMT) on SMT cores. The active thread executes the program and puts its results on a delay buffer. The redundant thread executes the same instruction stream and compares the results that it produces with the ones from the delay buffer. The committed state of the redundant thread is also used as a recovery checkpoint.

Several works are based on AR-SMT and extend it. [21] introduces Simultaneous and Redundant Threaded (SRT) processors that take advantage of an SMT processor's extra thread contexts. Similarly, [19] uses the SMT idea on CMPs proposing Chip-level Redundant Threading (CRT). [12] and [27] present techniques that exploit the idle cores for redundant thread execution. The main disadvantage of redundant multi-threading is that it reduces the system's total throughput because it occupies more thread contexts and hardware resources. Additionally, compared to compiler-based approaches, it requires custom hardware.

Process level redundancy (PLR) [25] replicates the processes of the application and compares their outputs to ensure correct execution. The processes synchronize to compare their outputs when the value escapes user space to the kernel. RAFT [9] improves this scheme by removing the synchronization barriers. PLR has small overhead since it checks fewer values than other approaches, but this comes at the cost of maintaining multiple memory states.

Wang [31] introduced **symptom-based** error detection. The main idea is that transient errors generate symptoms like memory exceptions, cache misses, branch mis-predictions etc. These symptoms can be used for error detection. In Shoestring [7], the error detection is based on symptoms, requiring less replication. This leads to better performance, but worse fault-coverage.

In **hardware** error detection, correctness is checked on hardware. Hardware-based designs include the watchdog processors in [16] and [3]. The main idea is that a smaller and simpler in design processor, which is considered safer, follows the execution of the main processor. Commercial processors like IBM's S/390 [26] replicate the entire execution unit.

7 Conclusion

We presented DRIFT, the first work that explores and solves a significant performance limitation in single-core error detection methodologies, namely, basic-block fragmentation. DRIFT is based on the idea of decoupling which breaks the execute-check-confirm-execute synchronization cycle existing in synchronized schemes. DRIFT decouples the execution of the code from the checks, resulting in code that the scheduler can optimize better as it is no longer limited by the complex control flow caused by the frequent checking. Our evaluation on a real machine shows significant performance improvements up to 29.7% and average performance overhead of 1.29× compared to native, non-fault tolerant, code. The performance gains have no impact on the fault-coverage compared to synchronized schemes.

References

1. GCC: GNU compiler collection. *http://gcc.gnu.org.*

2. SKI, an IA64 instruction set simulator. *http://ski.sourceforge.net*.
3. T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *MICRO*, 1999.
4. D. Bernick et al. Nonstop advanced architecture. *DSN*, 2005.
5. J. Chang et al. Automatic instruction-level software-only recovery. *DSN*, 2006.
6. C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 2003.
7. S. Feng et al. Shoestring: Probabilistic soft error reliability on the cheap. *ASPLOS*, 2010.
8. J. Fritts et al. Mediabench II video: Expediting the next generation of video systems research. *SPIE*, 2005.
9. Y. Ghosh et al. Runtime asynchronous fault tolerance via speculation. *CGO*, 2012.
10. J. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 2000.
11. W.-M. W. Hwu et al. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 1993.
12. C. LaFrieda et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. *DSN*, 2007.
13. M. Li et al. Understanding the propagation of hard errors to software and implications for resilient system design. *ASPLOS*, 2008.
14. P. G. Lowney et al. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 1993.
15. S. Mahlke et al. Sentinel scheduling for vliw and superscalar processors. *ASPLOS*, 1992.
16. A. Mahmood et al. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 1988.
17. C. McNairy et al. Itanium 2 processor microarchitecture. IEEE Micro, 2003.
18. S. Michalak et al. Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 2005.
19. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. *ISCA*, 2002.
20. N. Oh et al. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 2002.
21. S. Reinhardt et al. Transient fault detection via simultaneous multithreading. *ISCA*, 2000.
22. G. Reis et al. SWIFT: Software Implemented Fault Tolerance. *CGO*, 2005.
23. E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *FTCS*, 1999.
24. P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. *DSN*, 2002.
25. A. Shye et al. Using process-level redundancy to exploit multiple cores for transient fault tolerance. *DSN*, 2007.
26. T. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 1999.
27. J. Smolens et al. Reunion: Complexity-effective multicore redundancy. *MICRO*, 2006.
28. D. Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 2009.
29. J. Srinivasan et al. The impact of technology scaling on lifetime reliability. *DSN*, 2004.
30. C. Wang et al. Compiler-managed software-based redundant multi-threading for transient fault detection. *CGO*, 2007.
31. N. Wang et al. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 2006.
32. Y. Zhang et al. DAFT: Decoupled Acyclic Fault Tolerance. *PACT*, 2010.