Instruction-Level Fault Tolerance Configurability

Demid Borodin, B.H.H. (Ben) Juurlink and Stamatis Vassiliadis
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Telephone: +31 15 2786623, Fax: +31 15 2784898.
E-mail: {demid,benj,stamatis}@ce.et.tudelft.nl

***Abstract*—Fault tolerance (FT) is becoming increasingly important in computing systems. FT features are based on some form of redundancy, which adds a significant cost to a system, either increasing the required amount of hardware resources or degrading performance. To enable a user to choose between stronger FT or performance, some schemes have been proposed, which can be configured for each application to use the available redundancy to increase either reliability or performance. We propose to have an instruction-level, rather than application-level, configurability of this kind, since some applications (for example, multimedia) can have different reliability requirements for their different parts. We propose to apply weaker (or no) FT techniques to the less critical parts. This yields a certain time or resource gain, which can be used to apply stronger FT techniques to the more critical parts, thereby, increasing the overall FT. We show how some existing FT techniques can be adapted to support instruction-level FT configurability, and how a programmer can specify the desired FT of particular instructions or blocks of instructions in assembly or in a high-level programming language. In some cases compiler can assign the FT level to instructions automatically. Experimental results demonstrate that reducing the FT of non-critical instructions can lead to significant performance gains compared to a redundant execution of all the instructions. The fault coverage of this scheme is also evaluated, demonstrating that it is very application-specific. For some applications the fault coverage is very admissible, but unacceptable for others.**

I. Introduction

The importance of fault tolerance (FT) of computing systems is increasing instantly nowadays [1]. This is a consequence of the technology trends which try to follow Moore's law in increasing chip density by decreasing feature size. Smaller feature size, greater chip density, and minimal power consumption lead to increasing device vulnerability to external disturbances such as radiation, internal problems such as crosstalk, and other reliability problems, which result in an increasing number of faults, especially transients, in computing systems.

After the switch from tubes to more reliable transistors and until recently, a strong FT used to be a requirement only of special-purpose high-end computing systems. The technology reliability was considered sufficient, and only a few FT techniques, such as Error Correcting Codes (ECC) [2] in memory, were usually used. Already now, and, according to predictions, more and more in the (near) future, the technology trends pose the reliability problem [1], which leads to the need of FT features even in PCs.

This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

Many fault tolerant schemes exist. There is always a trade-off between FT and cost, either in performance or resources. System resources are limited, and the more of them are dedicated to FT, the more performance suffers. It is desirable, having a certain system, to be able to choose in every particular case if its resources should be used to improve FT or performance. Some proposed FT schemes may enable system configuration before an application is run, which allows to choose between higher performance or stronger FT depending on the application requirements.

We propose a system configurability targeting either FT or performance at the instruction, rather than application level. This is based on the observation that, for example, for multimedia applications, most of the computations do not strictly require a strong FT, because many errors would not be visible for a human, while others can cause a slight, tolerable inconvenience. However, if the control of a multimedia application is damaged, the application is likely to crash. Hence, the control instructions should preferably be highly fault tolerant, while the non-critical computations can be left as they are, or can be protected with weaker (and cheaper) techniques to avoid excessive redundancy which degrades performance and/or increases the system cost. Alternatively, the gained time or resources can be used to enhance the FT of the critical instructions even further. In this case, the overall reliability of the application increases, at the expense of reduced reliability of non-critical parts.

We call the strength of FT features applied to an instruction the **degree of FT**. The more efficient FT techniques are applied, the higher the degree of FT is. The minimum degree of FT corresponds to the absence of any FT techniques. Duplication and comparison of the results has a lower degree of FT than Triple Modular Redundancy (TMR) [3], [4]. Normally, a higher degree of FT corresponds to a greater amount of redundancy, and hence, is more expensive in terms of resources and/or time.

The proposed technique is referenced to as **Instruction-Level Configurability of Fault Tolerance (ILCOFT)**. A programmer is able to specify which instructions are critical,

and thus should be highly fault tolerant, and which instructions are not. If a system supports several degrees of FT, the programmer is able to specify the desired degree for each instruction or group of instructions. This can be done either in high level language or in assembly code. Partially, it could also be performed automatically by the compiler. The system adapts one of the existing FT schemes to satisfy the needs of particular instructions, for example, by duplicating or triplicating them in software or hardware, and comparing the results.

This paper is structured as follows. Section II discusses related work. Section III presents ILCOFT, giving its reasoning in a greater detail in Section III-A, showing how several existing FT schemes can be adapted for ILCOFT in Section III-B, and discussing possible ways for a programmer to specify the desired degree of FT for particular instructions or code blocks in Section III-C. Section IV demonstrates some experimental results. Finally, Section V draws conclusions and discusses future work.

II. Related Work

Saxena and McCluskey [5] noticed that multithreaded FT approaches can target both high-performance and high-reliability goals, if they allow configuration to either high-throughput or fault tolerant modes. For example, slipstream processors [6], [7] provide this configurability.

Breuer, Gupta, and Mak [8] proposed an approach called **error tolerance**, which increases the fabrication yield, by accepting fabricated dies which are not completely error-free, but deliver acceptable results. Among other considerations, the tolerance of multimedia applications to certain errors is discussed.

Chung and Ortega [9] developed a design and test scheme for the motion estimation process. This reveals that the effective yield can be improved if some faulty chips are accepted.

Lu [10] presents the Structural Integrity Checking technique using a watchdog processor [11] to verify the correctness of an application control flow. "Labels" are inserted into the application at the places where a check should be performed. The higher density of the "labels" is, the more checking is done. Thus, a programmer can increase the density of the "labels" at the critical parts of an application, increasing the amount of checking applied to them.

We propose to leverage the error tolerance of certain applications to improve their overall reliability and/or improve their performance. This goal is achieved by enabling instruction-level degree of FT configurability, which means that an application developer is able to configure the degree of FT applied to particular instructions or blocks of instructions. By reducing the degree of FT for non-critical and increasing it for critical instructions, a developer can improve the overall application reliability and/or performance.

III. ILCOFT

A. Motivation

Many multimedia applications, such as image, video and audio coders/decoders, use lossy algorithms. After decoding, a stream produced is not perfect. It incorporates errors which the human eye cannot notice or can easily tolerate. For example, if one of more than 307 thousand (640×480) pixels in an image or a video frame has a wrong color, it is likely to be ignored by a human. If an error occurs in calculations associated with motion compensation in video decoding, this can result in a wrong (rather small) block for one or a few frames. The number of frames that can be affected depends on the place where the error appeared and on how far the following key frame is. Because usually there are 20 to 30 frames per second, the chance that a human will notice this error is quite low. Moreover, if it is noticed, it will probably result in less inconvenience than the compression-related imperfections.

This shows that some errors can be allowed in this kind of applications. However, if an error occurs in the control part of a multimedia application, it is very likely that the whole application will crash.

As an example, consider the image addition kernel presented in Figure 1. If an error occurs in any of the expressions that evaluate the pixel value *sum*, it will result in a wrong pixel in the output image, which is tolerable. However, if a problem appears in the statements controlling the loops, there is a very small chance that it will not crash the application or seriously damage the results. A normal termination with correct results can happen in this case if one or both loops performed too many iterations, but the memory which they damaged was not used for any other purpose. This scenario, however, has a very low probability. It is likely that the application will crash (due to a jump to an invalid address, damage of memory, etc.), or, if the loop is exited too early, the part of the image which has not been processed yet will be wrong. The *if* statement which controls saturation is less dangerous than the loops, because if the condition is evaluated incorrectly, only one pixel suffers. If the branch target address is corrupted, however, the application will most probably crash. Thus, this *if* statement can also be considered for a higher degree of FT.

```
for( i=0; i<N; i++ )
    for( j=0; j<M; j++ )
    {
        sum = ImageX[i][j] + ImageY[i][j] ;
        if( sum > 255 )  /* saturation */
            sum = 255;
        ImageX[i][j] = sum;
    }
```

Fig. 1. Image addition

Hence, for the image addition kernel presented in Figure 1, it is desirable to have the maximum level of FT for the instructions controlling the loops and the branch target address of the *if* statement. A lower level of FT for the other

instructions, which determine the pixel value, is acceptable, and even desirable, when aiming at performance.

FT is based on some form of redundancy. Space redundancy, which increases the amount of required hardware resources, can achieve FT without a performance loss, at the expense of increased hardware cost. The amount of hardware is usually limited, however, and to achieve FT under this constraint, time redundancy is used, which degrades performance. When both hardware resources and time are limited, which is very common, the proposed scheme increases performance at the expense of decreased reliability of non-critical application parts. However, the critical parts are still as reliable as with a full FT scheme, so the overall application reliability is not affected. As Section IV-A shows, ILCOFT can provide a significant performance improvement.

B. FT Schemes adaptable to ILCOFT

We propose that a system can provide several FT techniques of varying strengths, corresponding to different degrees of FT. For example, a non-redundant instruction execution has FT degree *0* (no FT), duplication with comparison of the results can be assigned FT degree *1*, and a Triple Modular Redundancy (TMR) is associated with FT degree *2*.

Duplication and triplication of an instruction assumes either hardware or time redundancy, which results either in multiple execution units where the copies of an instruction can be executed simultaneously, or in a sequential (or partially sequential) execution of the multiple copies.

There exist several techniques for high-performance processors that try to minimize the effect which this created redundancy has on performance, and we show how they can be adapted to support ILCOFT.

For instance, a pure software technique named Error Detection by Duplicated Instructions (EDDI) [12] has been proposed, which takes advantage of Instruction Level Parallelism (ILP) [13] of superscalar processors to minimize the performance impact of the redundancy. This technique duplicates all the instructions and memory in software, using different registers. A software tool performs duplication automatically. Applying the proposed ILCOFT to this scheme assumes that a programmer can specify the required FT degree of different instructions. Without modifications, EDDI can support only two degrees of FT: *0* (no redundancy) and *1* (duplication and comparison). However, EDDI can be extended to allow more redundancy: triplication with voting etc. While compiling, each instruction is multiplicated according to its FT degree, then the results are compared or voted. In Section IV EDDI is adapted to support ILCOFT, with two possible degrees. It is shown using several multimedia kernels that minimizing the degree of FT for non-critical instructions provides a substantial performance gain.

If the FT techniques are implemented in hardware, there must be a way to set the required FT mode for every instruction. For example, several bits in the instruction encoding can specify the required FT degree. The number of bits allocated for this purpose depends on the number of available FT modes supported by hardware.

Franklin [14] proposed to duplicate instructions in superscalar processors at run time and compare the results to detect errors. Two places where instructions can be duplicated were presented and analyzed: (1) in the dynamic scheduler after an instruction is decoded, and (2) in the functional unit where the instruction is executed. To adapt this scheme to support ILCOFT, several bits in the instruction encoding can be set by the compiler to determine the needed FT degree for that instruction. Based on this information, the hardware performs the appropriate FT action, e.g., duplicate the instructions and compare their results, triplicate and vote, etc. This can be also applied to the scheme proposed in [15].

The DIVA approach [16], [17], [18] uses a simple and robust processor, called DIVA checker, to verify the operation of the high-performance speculative core. This approach can also be adapted to support ILCOFT by selecting the instructions whose results have to be verified by the DIVA checker.

ILCOFT is also applicable to FT techniques based on simultaneous multithreading [19], such as those presented in [5], [20], [21], [22], [23], slipstream processors [6], and others.

C. Specification of the Desired FT Degree

Two possible ways how a programmer can specify the desired degree of FT applied to an instruction are to set it in assembly code or in high-level language. Alternatively, in some cases, the compiler can perform this automatically.

Since a programmer most likely does not dream to mark the degree of FT for every assembly instruction or high-level language statement manually, the first step is to choose the appropriate policy which determines the default degree of FT, which is applied to all unmarked instructions. The default can be set to, for example, the minimum, average, or maximum possible degree of FT.

The approach which sets the default degree of FT to the minimum requires a programmer to mark instructions/statements that should receive a higher degree of FT. This method does not look very practical, because there is a high chance that many instructions are critical for an application: an illegal branch in any place can crash the whole application.

The opposite approach, when the default degree of FT is the maximum, looks more useful for many applications. In this case, a programmer marks the instructions or statements that should have the lower degree of FT, and all the others get a higher degree. This is especially suitable for multimedia applications, many of which spend most of the runtime in small kernels. Decreasing the degree of FT of a few computational instructions in a heavily used kernel can provide a significant application-level performance gain.

Finally, the default degree of FT can be assigned some intermediate value. Then, a programmer has to specify instructions/statements requiring higher and lower degree of FT.

1) In Assembly Code: If a programmer specifies the desired degree of FT in assembly code, the way how it can be done depends on the FT scheme which is used.

If FT is implemented in hardware, the programmer marks instructions with the required degree of FT using some flags, and the assembler encodes this information into every instruction.

In pure software, the EDDI technique [12] discussed in Section III-B can be used. This technique duplicates all the instructions and compares important results to detect possible errors. Adapting EDDI, a programmer can duplicate the critical instructions manually, taking care about the register allocation, register spilling, possibly memory duplication, etc. However, some automatic assisting tool would be very useful. This tool can be based on the compiler postprocessor used in [12], which automatically includes EDDI into an application. A compiler reserves registers for duplicate instructions, and a tool duplicates everything. In the resulting assembly file, a programmer removes the undesired redundancy manually.

2) In High-Level Language: Figure 2 demonstrates a possible way for a programmer to specify the desired degree of FT for particular statements or blocks of statements in a high-level language. This is done in the form of a *#pragma* statement which determines the degree of FT that should be applied to the following statements, until the next *#pragma* statement changes it. The larger the number corresponding to *FT_DEGREE* is, the higher degree of FT should be. Each statement is compiled into instruction(s) whose degree of FT is equal to that of the corresponding statement. In the case of control statements, a compiler must be able to find their dependencies and to apply the same degree of FT to them.

In Figure 2, the instructions which are generated for the *for* statement, should have the degree of FT equal to 3. The instructions inside the loop (and after the loop until the next *#pragma*) should have the degree of FT 1. Thus, the correct loop control is more critical than the addition result, which is typical for multimedia applications. Obviously, the loop control depends on the values of the variables i and n, which have been assigned before. Hence, the compiler should walk backwards to find all the instructions on which the values of these variables depend, and assign the degree of FT 3 to them.

```
#pragma FT_DEGREE 3

for( ; i < n; i++ )
{
#pragma FT_DEGREE 1

      c[i] = a[i] + b[i];
}
```

Fig. 2. Possible FT degree specification in a high-level language

3) Automatically by the Compiler: If a system supports only two degrees of FT, for example, *no FT* (no FT techniques are applied) and *fault tolerant* (some techniques are applied), in some cases the compiler can determine the instructions that need to be fault tolerant automatically. This saves a programmer from manual work. The automatic compiler scheme can be based on the observation that in most cases, the instructions on which an application's control flow depends, require a higher degree of FT. All control flow instructions, such as branches, jumps, and function calls, are assigned a higher degree of FT. Furthermore, all instructions on which these control flow instructions depend should also receive the higher FT degree. The efficacy of this scheme depends on the compiler's ability to perform exact dependence analysis. In the worst case, all instructions on which a control flow instruction could depend need to be given the higher FT degree.

IV. Experimental Results

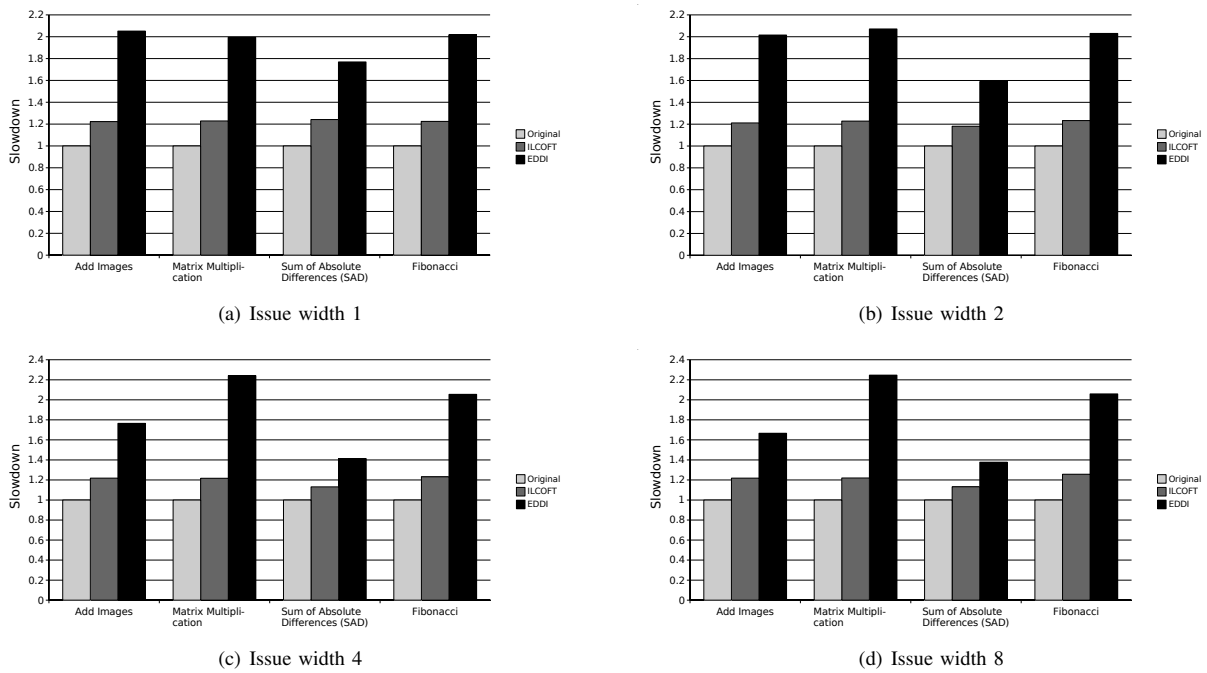
A. Performance Evaluation

To evaluate the performance gain delivered by applying ILCOFT, we adapt the EDDI [12] scheme to support ILCOFT. Performance results of four kernels in non-redundant (original), EDDI and ILCOFT-enabled EDDI forms are compared. The SimpleScalar simulator tool set [24], [25] is utilized for performance simulation. The default SimpleScalar PISA architecture is used.

Four simple kernels are investigated: image addition (matrix addition) discussed in Section III-A, matrix multiplication, sum of absolute differences (SAD), and a Fibonacci numbers generator. For each kernel, the C source code is compiled to SimpleScalar assembly code. The compiler-optimized (GCC *-O2* flag) version of the application plays the role of the "original", non-redundant application, with no FT.

The EDDI version of the kernel is derived from the original version by duplicating all the instructions and memory, and integrating the checking instructions, by hand, according to the scheme presented in [12]. Memory duplication means that the data memory has a shadow copy which is referenced by the duplicate instructions. Thus, after any duplicate instruction has finished its execution, the contents of a duplicate of a memory structure must always be equal to that of the original structure. Inequality signals an error. Checking instructions only appear before a value is stored or used to determine a conditional branch outcome. Faults are free to propagate within intermediate results. This is proposed in [12] to minimize the performance overhead.

The ILCOFT-enabled EDDI version is obtained from the original application by duplicating only the critical instructions in the kernel and comparing their results, without memory duplication. Most of the control instructions are considered to be critical. For the image addition kernel, these are the instructions to which the loop control statements in Figure 2 are compiled, and the instructions on which the control variables depend. Memory duplication does not make sense for ILCOFT-enabled EDDI, because all the instructions have to be duplicated to maintain a shadow memory copy. In our experiments, the applications FT does not suffer from this, because the control instructions do not depend on memory contents, but only on register values, which are protected.



(a) Issue width 1

(b) Issue width 2

(c) Issue width 4

(d) Issue width 8

Fig. 3. Slowdown of EDDI and ILCOFT-enabled EDDI over non-redundant kernels, for varying issue widths

Both EDDI and ILCOFT-enabled EDDI reliably protect only against transient hardware faults that do not last longer than one instruction execution. To protect against faults taking more time, including permanent faults, there should be a way to ensure that an instruction and its duplicate execute on different hardware units. For example, they can execute on different CPUs in a multicore system, or on different functional units of a superscalar processor. In the latter case, only long-lasting faults of functional units are covered. Alternatively, to avoid hardware replication, techniques changing the form of the operands of a duplicate instruction, such as alternating logic [26] and recomputing with shifted operands [27], can be used. However, they are expected to have a significant impact on performance, and are outside the scope of this paper.

Figure 3 depicts the slowdown of EDDI and ILCOFT-enabled EDDI over the non-redundant scheme for four different processor issue widths. Figure 4 demonstrates the ratio of the number of committed instructions of both schemes to that of the non-redundant scheme. Without ILP, speculation etc., Figure 3 is expected to be similar to Figure 4. The performance results of Figure 3(a) (issue width 1) are quite consistent with Figure 4, but for larger issue widths, the processor exploits the available parallelism better (the original instruction and its duplicate are independent). Because of this, the slowdown of EDDI and ILCOFT-enabled EDDI decreases when the issue width increases, unless there are other limiting factors. Matrix multiplication, for example, has a structural hazard: there is only one multiplier, so the duplicate of a multiplication instruction cannot be executed in parallel with the base instruction.

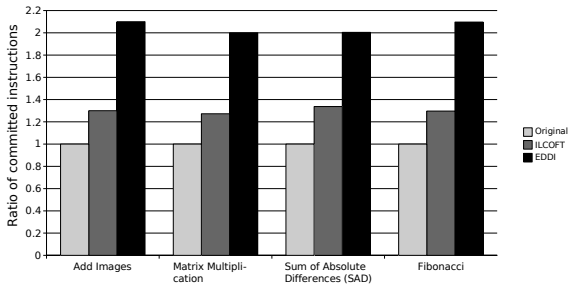


Fig. 4. Committed instructions

As in [12], despite duplication of all instructions and memory in EDDI, especially for larger issue widths, its slowdown over the original application is in most cases smaller than the intuitively anticipated two times (actually, more than two because of the checking instructions, duplicated memory, and register spilling). This is due to the increased ILP introduced by the duplicates which are independent of the original instructions. This leads to a more efficient resource usage and fewer pipeline stalls. ILCOFT-enabled EDDI also profits from this feature.

Figure 3 shows that ILCOFT-enabled EDDI is considerably faster than EDDI for all kernels. Several factors contribute to this:

- The number of instructions in ILCOFT-enabled EDDI is smaller than in EDDI (by about 40% on average).
- EDDI duplicates memory, while ILCOFT-enabled EDDI does not.

- EDDI needs more registers than ILCOFT-enabled EDDI, since ILCOFT-enabled EDDI duplicates fewer instructions and, hence, reduces register pressure. Higher register usage leads to more register spilling.

As these factors have different weights for different kernels, the speedup of ILCOFT-enabled EDDI over EDDI is not constant. For the image addition kernel, memory duplication contributes 1.3% to the speedup of ILCOFT-enabled EDDI over EDDI. The contribution of additional register spilling (two more registers are saved in stack for EDDI) is negligible (less than 1%). The remaining contribution should be attributed to the duplicated instructions.

B. Fault Coverage Evaluation

In this section we provide an initial evaluation of the fault coverage of ILCOFT-enabled EDDI. The purpose is to determine how ILCOFT affects the fault coverage of EDDI.

We simulate hardware faults by extending the SimpleScalar *sim-outorder* simulator with a fault injection capability. At a specified frequency (every N instructions) a fault is injected by corrupting an input or output register of an instruction (overwriting its content with a random value). Only integer arithmetic instructions are affected, because only this kind of instructions appears in the tested kernels, in addition to memory access and branch instructions. The faults inside memory access and branch instructions are not covered by EDDI (only their inputs are protected), thus ILCOFT-enabled EDDI is also not expected to cover them, so they are not affected by the fault injector. Fault injection into an input register simulates a memory, bus or register file fault. Fault injection into an output register simulates a functional unit fault also. Faults are injected only within the kernel code, because the main function is not protected in our experiments.

We remark that the fault appearance does not represent a realistic model. The aim here is to evaluate the behavior (fault coverage) of the investigated schemes under different fault pressures (frequencies), and to ensure that as many as possible of the fault propagation paths within the kernels are examined. By making the fault injection periodic rather than random, and by varying the frequency for each of a large number of simulations, we attempt to gain a better control over the process, and to achieve the mentioned goals.

Table I, Table II, and Table III present the faults injection results for the three different schemes. The first column of each table shows the number of simulations executed. The chosen number of simulations differs for each kernel, and depends on the number of committed instructions. The frequency of injected faults starts from one fault per every 1000 (in some cases 100) instructions, and every new simulation decreases the fault frequency until it becomes roughly one fault per execution. In this way we make sure that all the situations with frequent down to rare faults are evaluated, and that random instructions within kernels are affected. The second column shows how often faults have been detected by the FT scheme (EDDI or ILCOFT-enabled EDDI), the third column – faults detected by the simulator (for example, a memory access to an illegal address is reported). The fourth column contains the percentage of undetected faults, which shows how often the kernel execution finished without reporting an error. The fifth column demonstrates how often an application crashed, i.e., did not produce any output. The sixth column shows how often an application delivered a correct result despite the presence of undetected fault(s). In other words, the application has finished execution without reporting an error, but a certain number of faults occurred which have not been detected. Nevertheless, the application output was correct, since the faults have not propagated to it. In parentheses the maximum number of undetected faults in this situation is given. The seventh column gives the maximum number of faults injected per execution, before the execution finished normally or was interrupted reporting an error. There are usually fewer injected faults in EDDI than in other schemes, because EDDI detects and reports faults, aborting the execution, earlier. The next column shows the maximum number of undetected faults, which were injected but not detected, and the execution finished without reporting an error. Most of the times these undetected faults result in corrupted application output, except the cases counted in the sixth column. Finally, the ninth column presents the maximum, and the last column – the average output corruption caused by undetected faults, demonstrating how many of them propagate to the output. An output corruption percentage is defined as a ratio of the number of wrong to the number of correct application outputs. The average output corruption is calculated as a sum of all the corruption percentages divided by the number of simulations, and is used to emphasize that a very high maximum output corruption does not necessarily mean that the output is usually corrupted by the same amount. It can be an exceptional case.

Obviously, ILCOFT affects kernels in very different ways. The difference in the fault coverage can be explained by the density of the duplicated instructions in a kernel. The more instructions are duplicated, the better fault coverage is, and the lower performance gain is. Among the presented kernels, the worst fault coverage (the greatest percentage of undetected faults) appears in matrix multiplication and sum of absolute differences, in which relatively many unprotected computational instructions reside between the protected control instructions. Depending on the application, the significant performance increase at the expense of the weak fault coverage can be considered acceptable. For example, for sum of absolute differences used in motion estimation, a wrong result leads to a wrong block, which can usually be tolerated.

The exceptionally high percentage of correct outputs despite faults in matrix multiplication can be explained by the fact that most of the results (output matrix elements) are truncated when overflow occurred. The truncation masks many errors. This can also be one of the reasons why matrix multiplication has a relatively small percentage of detected faults: the faults are masked before they propagate to a checking instruction which can detect them. With a higher calculations precision (more bits per value), the number of correct outputs despite faults would drop.

TABLE I
FAULT INJECTION RESULTS FOR THE ILCOFT SCHEME FOR THE FOLLOWING KERNELS: IMAGE ADDITION (IA), MATRIX MULTIPLICATION (MM), FIBONACCI NUMBERS GENERATION (FIB) AND SUM OF ABSOLUTE DIFFERENCES (SAD)

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Correct output despite faults % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	13526	86.66	0	13.34	0	0.07 (2)	22	11	0.13	0.012
MM	621	53.62	0	46.38	0	23.19 (5)	15	11	99	3.103
Fib	581	66.44	25.99	7.57	0	0	8	3	96.67	38.232
SAD	340	55.88	0	44.12	0	0.29 (1)	25	18	100	100

TABLE II
FAULT INJECTION RESULTS FOR THE EDDI SCHEME

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Correct output despite faults % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	6025	100	0	0	0	0	2	0	0	0
MM	621	100	0	0	0	0	11	0	0	0
Fib	581	67.81	32.01	0.17	0	0	3	2	96.67	96.667
SAD	340	98.53	0	0.29	1.18	0.29 (1)	23	1	100	100

TABLE III
FAULT INJECTION RESULTS FOR THE NON-REDUNDANT SCHEME

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Correct output despite faults % (max. # faults)	Max. # injected faults	Max. # undetected faults	Max. output corruption %	Av. output corruption %
IA	2768	n/a	0	100	0	0	6438	6438	99.66	1.074
MM	621	n/a	0	100	0	43 (9)	50	50	94.75	2.992
Fib	532	n/a	32.33	67.67	0	10.71 (4)	5	5	97.78	53.991
SAD	326	n/a	0	100	0	8.28 (10)	130	130	100	100

The most important fault coverage characteristic from a user point of view is the final output corruption. The fact that a certain amount of corruption can be allowed in some applications drives the idea behind ILCOFT. Obviously, this is a very application-specific issue, depending solely on the algorithm employed.

The image addition kernel, computing every pixel value independently, without a long chain of computations, shows a very good result: only a few pixels (maximum 0.13% of the whole output image) are corrupted. This can often be unnoticed by a user. The maximum output corruption happened when a fault was injected into the register which held the base address of an array representing one input image line (matrix row), and was later used to fetch all the image data on this line. As a result, garbage was fetched from a random memory location for every pixel of the rest of the line, and the resulting image line was entirely corrupted from the point where fault appeared. This was quite visible on the output image. It could be solved by performing checks of computed addresses before every load and store. Then, only single pixels would have been affected.

In all other kernels, the resulting values depend on a long chain of computations, and even on each other, so the final output corruption increases dramatically. In Fibonacci numbers generation, every subsequent value depends on the previous one, so, all the values behind the first erroneous one become wrong. This leads to the unexpected extremely high final output corruption in EDDI, which catches an eye in Table II. However, in fact, only one of 581 simulations finished with an undetected error (2 undetected faults), and this error obviously manifested among the first Fibonacci numbers, so all the following numbers were computed on the base of this error, and thus, about 97% of the final output was corrupted. The average output corruption of about 97% is equal to the maximum, because this is the only undetected error. Sum of absolute differences delivers only one value as a result, which can be either correct or wrong, and any unmasked fault in the computations leads to an error. Consequently, all the undetected errors affect 100% of the output.

The experimental results demonstrate that the fault coverage of ILCOFT-enabled EDDI can be significantly improved by protecting the computed memory access addresses. For example, as mentioned, it could solve the corrupted output line problem in image addition. This can be done before every load and store instruction, by checking the value of the register which holds the memory address. Of course, the redundant value must be computed by a chain of duplicated instructions (which can be done automatically by a compiler), bringing back the trade-off between performance and fault coverage.

The memory access address problem is not relevant for EDDI, because the memory is duplicated there, and all the loads and stores reference different memory locations. However, this can be a point where the fault coverage of ILCOFT-enabled EDDI is stronger than that of EDDI itself: EDDI does not have any memory address protection, so a fault in a store can damage any memory location, while ILCOFT-enabled EDDI with memory access protection saves from this.

To minimize the performance loss, only the store addresses can be protected, assuming that a memory corruption is worse than fetching a wrong value. But in this case, the image addition corrupted line problem discussed above is not solved.

V. Conclusions

In this work we have proposed an instruction-level, rather than application-level, configurability of FT techniques applied to an application. This idea is based on the observation that some applications might pose different FT requirements for their different parts. For example, in multimedia applications, an error in parts calculating the value of a pixel, a motion vector, or a sample frequency (sound) can be easily unnoticed or ignored by a human observer. However, an error in the control (critical) part will most probably lead to a crash of the whole application. This suggests that it is most important to apply the strongest FT features to the critical parts, and non-critical parts can be protected with a weaker FT to improve the application performance. In applications with execution time constraints, the time saved by reducing the FT of non-critical parts can be used to further increase the FT of the critical parts, thus improving the overall application reliability.

We have shown how several existing FT schemes can be adapted to support ILCOFT, proposed a way how a programmer can specify the desired degree of FT in a high-level language or assembly code, and indicated how a compiler can apply FT techniques to control code automatically.

The experimental results have demonstrated that ILCOFT is able to significantly improve an application performance when applying a higher FT degree to its critical parts (instructions) only. They have also shown that fault coverage of ILCOFT is very application-specific and works best with applications that compute independent elements. The fault coverage certainly depends on the amount of redundancy applied. Finally, we have demonstrated that adding memory access address protection can significantly improve the fault coverage.

Future work consists of applying ILCOFT to other FT schemes, also in hardware. Furthermore, development of compiler support for specification of FT degree is necessary to evaluate ILCOFT for large applications, such as audio/video codecs.

References

[1] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *DSN-02: Proc. 2002 Int. Conf. on Dependable Systems and Networks*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society Press.

[2] T. R. N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[3] J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 43–98. Princeton University Press, Princeton, NJ, 1956.

[4] B.W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Jan 1989.

[5] N.R. Saxena and E.J. McCluskey. Dependable Adaptive Computing Systems–the ROAR project. In *Proc. IEEE Systems, Man, and Cybernetics Conf*, volume 3, pages 2172–2177, Oct 1998.

[6] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[7] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *MICRO-33: Proc. 33rd annual ACM/IEEE Int. Symp. on Microarchitecture*, pages 269–280, New York, NY, USA, 2000. ACM Press.

[8] M.A. Breuer, S.K. Gupta, and T.M. Mak. Defect and Error Tolerance in the Presence of Massive Numbers of Defects. *IEEE Design and Test of Computers*, 21(3):216–227, 2004.

[9] Hyukjune Chung and Antonio Ortega. Analysis and Testing for Error Tolerant Motion Estimation. In *DFT-05: Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 514–522, Washington, DC, USA, Oct 2005. IEEE Computer Society Press.

[10] David Jun Lu. Watchdog Processors and Structural Integrity Checking. *IEEE Transactions on Computers*, C-31(7):681–685, Jul 1982.

[11] A. Mahmood and E.J. McCluskey. Concurrent Error Detection Using Watchdog Processors–A Survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[12] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, Mar 2002.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann, third edition, May 2003.

[14] Manoj Franklin. A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 207–215, Nov 1995.

[15] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. *MICRO-34*, pages 214–224, Dec 2001.

[16] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO-32: Proc. 32nd annual ACM/IEEE Int. Symp. on Microarchitecture*, pages 196–207, Washington, DC, USA, Jun 1999. IEEE Computer Society Press.

[17] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient Checker Processor Design. In *MICRO-33: Proc. 33rd annual ACM/IEEE Int. Symp. on Microarchitecture*, pages 87–97, New York, NY, USA, 2000. ACM Press.

[18] Chris Weaver and Todd Austin. A Fault Tolerant Approach to Microprocessor Design. *Dependable Systems and Networks*, pages 411–420, Jul 2001.

[19] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA-95: Proc. 22nd annual Int. Symp. on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM Press.

[20] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS-29*, pages 84–91, Madison, Wisconsin, USA, Jun 1999. IEEE Computer Society Press.

[21] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *ISCA-00: Proc. 27th annual Int. Symp. on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM Press.

[22] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *ISCA-02: Proc. 29th annual Int. Symp. on Computer architecture*, pages 87–98, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *ASPLOS-VII: Proc. seventh Int. Conf. on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM Press.

[24] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[25] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[26] Reynolds and G Metze. Fault Detection Capabilities of Alternating Logic. *IEEE Transactions on Computers*, C-27(12):1093–1098, Dec 1978.

[27] J.H. Patel and L.Y. Fung. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *IEEE Transactions on Computers*, C-31(7):589–595, Jul 1982.