J Sign Process Syst (2009) 57:89–105
DOI 10.1007/s11265-008-0175-9

Instruction-Level Fault Tolerance Configurability

Demid Borodin · B. H. H. (Ben) Juurlink · Said Hamdioui · Stamatis Vassiliadis

Received: 12 October 2007 / Revised: 4 January 2008 / Accepted: 5 March 2008 / Published online: 12 April 2008

Abstract Due to modern technology trends such as decreasing feature sizes and lower voltage levels, fault tolerance (FT) is becoming increasingly important in computing systems. Several schemes have been proposed to enable a user to configure the FT at the application level, thereby enabling the user to trade stronger FT for performance or vice versa. In this paper, we propose supporting instruction-level rather than application-level configurability of FT, since different parts of some applications (e.g., multimedia) can have different reliability requirements. Weak or no FT will be applied to less critical parts, resulting in time and/or resource gains. These gains can be used to apply stronger FT techniques to the more critical parts; hence increasing the overall reliability. The paper shows how some existing FT techniques can be adapted to support instruction-level FT configurability, how a programmer can specify the desired FT level of the instructions, and how the compiler can manage it automatically. A comparison between the existing FT scheme EDDI (which duplicates all instructions) and the proposed approach is performed both at the kernel and at full application levels. The simulation results show that both the performance and the energy consumption are significantly improved (up to 50% at the kernel and up to 16% at full application level), while the fault coverage depends on the application. For the full application (JPEG encoder), our approach is only applied to one kernel in order to avoid increasing the programming effort significantly.

Keywords Fault tolerance · Reliability · Performance · Energy consumption · Instruction-level configurability

D. Borodin (✉) · B. H. H. (Ben) Juurlink · S. Hamdioui · S. Vassiliadis
Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: demid@ce.et.tudelft.nl

B. H. H. (Ben) Juurlink
e-mail: benj@ce.et.tudelft.nl

S. Hamdioui
e-mail: said@ce.et.tudelft.nl

S. Vassiliadis
e-mail: stamatis@ce.et.tudelft.nl

1 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. 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, 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. However, according to [1, 3], the technology trends

will pose more and more reliability issues in future. This means, in turn, that FT features are required even in PCs.

Many fault tolerant schemes exist. There is always a trade-off between FT and cost, either in performance or resources. System hardware resources are limited, and the more of them are dedicated to FT, the more performance suffers. Furthermore, the redundancy introduced to provide FT dissipates additional energy. It is therefore desirable to have a configurable system which is able to use its resources to improve either 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.

Saxena and McCluskey [4] notice 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, as is the case for slipstream processors [5, 6]. Breuer, Gupta, and Mak [7] propose an approach called Error Tolerance, which increases the fabrication yield. This is achieved by accepting fabricated dies which are not completely error-free, but deliver acceptable results. The tolerance of multimedia applications to certain errors is discussed in this context. Chung and Ortega [8] develop 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. Reis et al. [9] present the software fault detection scheme *software implemented fault tolerance* (*SWIFT*) which duplicates instructions, compares their results at strategic places, and checks the control flow. The authors mention that SWIFT can allow a programmer to protect different code segments to varying degrees, like our scheme does. Oh and McCluskey [10] introduce the technique called *selective procedure call duplication* (*SPCD*), which minimizes the energy consumption and performance overhead of protective redundancy. For every procedure, SPCD either duplicates all its statements in the high-level language source code, or duplicates the call to the whole procedure, comparing the results. The decision is made based on the error latency constraints imposed. Procedure-level duplication is a coarse-grain version of fine-grain statement-level duplication. It reduces the energy and performance overhead by decreasing the number of checks executed. Experimental results show that SPCD provides the average energy savings of 26.2% compared to EDDI [11] (a software FT scheme which duplicates all the assembly instructions). However, SPCD does not guarantee protection against control-flow errors within a procedure whose call is duplicated. Lu [12] presents the Structural Integrity Checking technique using a watchdog processor [13] 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 the density of the "labels", 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 natural error tolerance of certain applications to improve their overall reliability and/or improve their performance and resource consumption. This goal can be achieved by a system which may be configured to target either FT or performance at the instruction, rather than application, level. A developer should be able to configure the strength of FT techniques applied to particular instructions or blocks of instructions in the application. This is useful for applications in which more and less critical parts (instructions) can be distinguished. For example, as noticed in [7, 8], for multimedia applications, most of the computations do not strictly require absolute correctness. Many errors in these computations would not be noticeable for a human, while others can cause a slight, tolerable inconvenience. The application parts performing these computations can have lower or no protection with a little risk. This minimizes protective redundancy which degrades performance and/or increases the system cost. However, other parts of the same applications can be very critical. For example, if the control of a multimedia application is damaged, the whole application is likely to crash. As another example, a fault in the data assignment to the quantization scale can affect the quality of the video significantly. These parts require a strong FT. Moreover, the time and/or resources saved by reducing redundancy for non-critical parts can be used to enhance the FT of the critical parts even further. In this case, the overall reliability of the application increases, at the expense of reduced reliability of non-critical parts. By reducing the protection of non-critical and increasing it for critical application parts, a developer can play with the trade-off between resources and reliability, fine-tuning it for the particular purposes.

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) [14, 15]. 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 referred to as *Instruction-Level Configurability of Fault Tolerance* (*ILCOFT*). If a system supports several degrees of FT, an application developer is able to specify the desired degree of FT 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 possibly comparing the results.

This paper is organized as follows. Section 2 presents ILCOFT. Section 3 demonstrates and analyzes experimental results at the kernel level, and Section 4—at the application level. Finally, Section 5 draws conclusions and discusses future work.

2 ILCOFT

This section presents *instruction-level configurability of fault tolerance* (*ILCOFT*). ILCOFT allows to apply different degrees of FT to different application parts, depending on how critical they are. ILCOFT is a general technique that can be applied to many existing FT schemes, as will be shown in Section 2.3. A particular ILCOFT implementation depends on the system architecture (the FT scheme used), and the application constraints. ILCOFT can be applied both to hardware and software FT schemes. A certain hardware support is required in the case of hardware FT schemes. Moreover, ILCOFT always requires a certain software-level activity to assign degrees of FT to application parts, as will be discussed in Section 2.2. The actions taken by the system in the case of a fault detection depend on the FT scheme adapted. For example, FT schemes providing only error detection will terminate the faulty execution unit, possibly perform a graceful degradation, etc. FT schemes supporting recovery may recover and continue execution. The FT characteristics of ILCOFT-enabled FT schemes depend on the FT techniques which are adapted and on the developer's instructions ranking in terms of their criticality.

Section 2.1 gives the reasoning behind ILCOFT. Section 2.2 discusses the possible ways for an application developer to specify the required degree of FT for particular instructions or code blocks. Section 2.3 shows how several existing FT schemes can be adapted to support ILCOFT.

2.1 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, it 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. Errors can be allowed in this kind of computations. However, if an error occurs in the control part of a multimedia application, it is very likely that the whole application will crash. Errors in some other parts can lead to a significant output corruption. Therefore, errors are not allowed to occur in the latter cases.

For yet more insight into the classification of critical and non-critical instructions, consider the image addition kernel presented in Fig. 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; this 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 (read) later. 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

```
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;
    }
```

Figure 1 Image addition.

which controls saturation is less critical 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.

In ILCOFT, the programmer specifies the required FT degree of every instruction or group of instructions. In other words, the programmer indicates which parts of an application are critical and which are not. In Section 2.2 we describe how this can be done by the programmer, and under which circumstances it can be performed automatically by the compiler. For example, for the image addition kernel presented in Fig. 1, the programmer should specify the maximum FT degree for the instructions controlling the loops and the branch target address of the *if* statement. For the other instructions, which calculate the pixel values, a lower FT degree is acceptable, and even desirable, when aiming at performance and resource consumption minimization.

By reducing the degree of FT of non-critical instructions, ILCOFT reduces the need in time or resource redundancy implementing FT. Space redundancy, which increases the amount of required hardware and energy resources, can achieve FT without a performance loss, at the expense of increased resources cost. The amount of hardware is often limited, however, and to achieve FT under this constraint, time redundancy is used, which degrades performance, and keeps energy consumption high. When both resources and time are limited, which is very common, ILCOFT increases performance and reduces energy consumption 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. Optionally, the saved time can be used to further improve the FT of the critical application parts by applying more time-redundant techniques to them. In this case the overall application reliability increases, because its critical parts are protected better.

2.2 Specification of the Required 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 the compiler can perform this automatically.

We do not consider it feasible for large applications that a programmer marks the required degree of FT for every assembly instruction or high-level language statement manually. It makes sense first to choose the appropriate policy which determines the default degree of FT. The default degree of FT is applied automatically to all unmarked instructions. It can be set to, for example, the minimum, maximum, or average possible degree of FT, as will be explained below.

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, e.g. 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 (we demonstrate this in Section 4).

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.

Next we discuss how an application developer can specify the degree of FT in the source code, and how it can be done automatically by the compiler.

2.2.1 In Assembly Code

If a developer specifies the required degree of FT in assembly code, the way how it can be done depends on the FT scheme which is used, if it is a hardware or software technique.

If FT is implemented in hardware, the way the programmer marks instructions might depend on how the degree of FT is passed to the hardware (see Section 2.3). With the degree of FT information embedded in instruction encoding, the programmer marks instructions using some flags, and the assembler encodes the necessary information into every instruction. With special FT mode configuration instructions, a programmer places these instructions in appropriate places. With separate versions of every instruction, the programmer uses an appropriate version. Alternatively, the assembler can be designed to support hardware-independent marking, which is translated automatically into the supported FT degree communication scheme. Then a programmer always marks instructions in the same way.

In pure software, the EDDI technique [11] discussed in Section 2.3 can be used. Adapting EDDI, a

programmer can duplicate the critical instructions manually, taking care about the register allocation, register spilling, possibly memory duplication, etc. However, an automatic assisting tool would be very useful. This tool can be based on the compiler postprocessor used in [11], which automatically includes EDDI into an application. A compiler reserves registers for duplicate instructions, and the tool duplicates everything. In the resulting assembly file, the programmer removes the undesired redundancy manually.

2.2.2 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 appropriate degree of FT to them. To be on the safe side, by appropriate degree of FT here we mean the highest between the previously assigned degree and the one required for the considered control statements.

In Fig. 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. 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];
}
```

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

2.2.3 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, for example, 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.

2.3 FT Schemes Adaptable to ILCOFT

Fault tolerant systems adapted to support ILCOFT need to 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 assumes either hardware or time redundancy. Hardware redundancy can be represented by multiple execution units where the copies are executed simultaneously. Time redundancy is provided by a sequential (or partially sequential) execution of multiple copies.

Because the main goal of ILCOFT is performance and energy consumption optimization, we focus on FT techniques that aim similar objectives. ILCOFT does not target systems for which only a high level of FT is important, and a large amount of redundancy is not an issue. There exist several techniques for high-performance processors that try to minimize the performance overhead created by protective redundancy. Below we discuss how some of them can be adapted to support ILCOFT.

Error detection by duplicated instructions (*EDDI*) [11] is a pure software technique. It duplicates all instructions in the program assembly code and inserts checks to determine if the original instruction and its duplicate produce the same result. More precisely, the registers are partitioned into two groups, one for the original instructions and one, called the shadow registers, for the duplicate instructions. After the execution

of a duplicate instruction, the contents of the shadow register(s) it affects should be identical to the contents of the destination register(s) of the original instruction. A mismatch signals an error. Instead of comparing the registers after the execution of every duplicate instruction, EDDI allows faults to propagate until the point where the value is saved to memory, and detects them just before saving. In other words, EDDI compares the registers only before their values are stored in memory. This minimizes the number of checking instructions needed, and thus, reduces the performance overhead, while data integrity is still guaranteed. EDDI also duplicates the data memory. This means that the data memory has a shadow copy which is referenced by the duplicate load/store instructions. Thus, after any duplicate store instruction, the contents of the shadow data memory must be the same as the original data memory.

From the point of view of performance overhead minimization, the main idea behind EDDI is that most applications cannot profit from wide-issue superscalar processors because they do not exhibit sufficient *instruction-level parallelism* (*ILP*) [16]. Because the original instructions and the duplicate instructions are independent, applying EDDI will increase ILP and, therefore, detect errors with a minimal or reasonable performance overhead in superscalar processors.

The original EDDI scheme supports only one degree of FT: duplication and comparison. It is straightforward, however, to extend EDDI to allow more redundancy, implementing, for example, triplication with voting. A lower degree of FT (no redundancy) can be easily achieved by avoiding duplication of certain instructions. From now on we assume that the user can specify the degree of replication.

In ILCOFT-enabled EDDI, only critical instructions are replicated. As discussed in Section 2.2, the programmer specifies the required FT degree of all program statements or assembly instructions. Alternatively, it is done automatically by the compiler. During compilation, each instruction is replicated according to its FT degree and then the results are compared or voted. Memory replication is not used in ILCOFT-enabled EDDI because all instructions have to be replicated to maintain a consistent memory copy. Instead, memory protection can be implemented by using ECC or other popular methods, preferably in hardware (this is outside the scope of this paper). In Sections 3 and 4, the performance and energy dissipation of ILCOFT-enabled EDDI is compared to those of EDDI. It will be shown that minimizing the degree of FT for non-critical instructions provides a substantial gain. Besides that, the fault coverage of these schemes will be evaluated.

Franklin [17] proposed to duplicate instructions in superscalar processors at run time and compare the results to detect errors. The 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, the required FT degree of executed instructions has to be passed to the hardware. Based on this information, the hardware performs the appropriate FT action, i.e., duplicates the instructions if necessary. This can be also applied to the scheme proposed in [18].

The DIVA approach [19–21] 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 [22], such as those presented in [4, 23–26], slipstream processors [5], and others.

It should be noted that for FT techniques 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. Alternatively, special instructions can be introduced which configure hardware to work in the desired FT degree mode. Finally, separate versions of each instruction can be created for every supported FT degree. The last solution does not look promising, however, because it implies a large overhead.

3 Kernel-Level Validation

This section presents experimental results with several kernels. The advantages and disadvantages provided by applying ILCOFT to an existing FT scheme are evaluated. Due to the experimental setup limitations, we work with only one FT scheme, i.e. the software error detection technique EDDI [11]. We adapt EDDI to support ILCOFT. EDDI and ILCOFT-enabled EDDI are presented in Section 2.3.

The FT features of EDDI and ILCOFT-enabled EDDI are discussed first. ILCOFT-enabled EDDI limits the sphere of replication of EDDI, protecting only the critical instructions, and avoiding memory duplication. 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 multiprocessor system, or on different functional units of a superscalar processor. In the latter case, long-lasting faults are covered only within the functional units. Alternatively, to avoid hardware replication, techniques changing the form of the operands of a duplicate instruction, such as alternating logic [27] and recomputing with shifted operands [28], can be used. However, these enhancements are expected to have a significant impact on performance, and are outside the scope of this paper.

The following kernels are investigated: *image addition* (*IA*), discussed in Section 2.1 (Fig. 1), *matrix multiplication* (*MM*), *sum of absolute differences* (*SAD*), and a *Fibonacci numbers generator* (*Fib*). Section 3.1 presents and analyzes the performance evaluation results, Section 3.2—the energy results, and Section 3.3—the fault coverage evaluation.

3.1 Performance Evaluation

To evaluate the performance gain delivered by applying ILCOFT to EDDI, performance results of four kernels in non-redundant (i.e. original), EDDI and ILCOFT-enabled EDDI forms are compared. The SimpleScalar simulator tool set [29, 30] is utilized for performance simulation. The default SimpleScalar PISA architecture is used.

For each kernel, the C source code is compiled to SimpleScalar assembly code. The compiler-optimized version of the application (i.e. compiled by GCC with *-O2* flag) 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 hand, according to the specification presented in [11]. All the instructions and memory are duplicated, and the checking instructions are integrated. 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 [11] 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. The original to ILCOFT-enabled EDDI transformation is also performed by hand. The control instructions are considered to be critical. For the IA kernel, these are the instructions to which the loop control statements in Fig. 2 are compiled, and the instructions on which the control variables depend.

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., Fig. 3 is expected to be similar to Fig. 4. The performance results of Fig. 3a (issue width 1) are quite consistent with Fig. 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. MM, 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.

Figure 3 shows that 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 happens due to the increased ILP introduced by the duplicates which are independent on 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 also shows that ILCOFT-enabled EDDI is considerably (up to 50%) faster than EDDI. Several factors contribute to this:

- The number of instructions in ILCOFT-enabled EDDI is smaller than in EDDI (by about 40% on average, see Fig. 4).
- 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 example, for the IA kernel, the simulation results show that 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

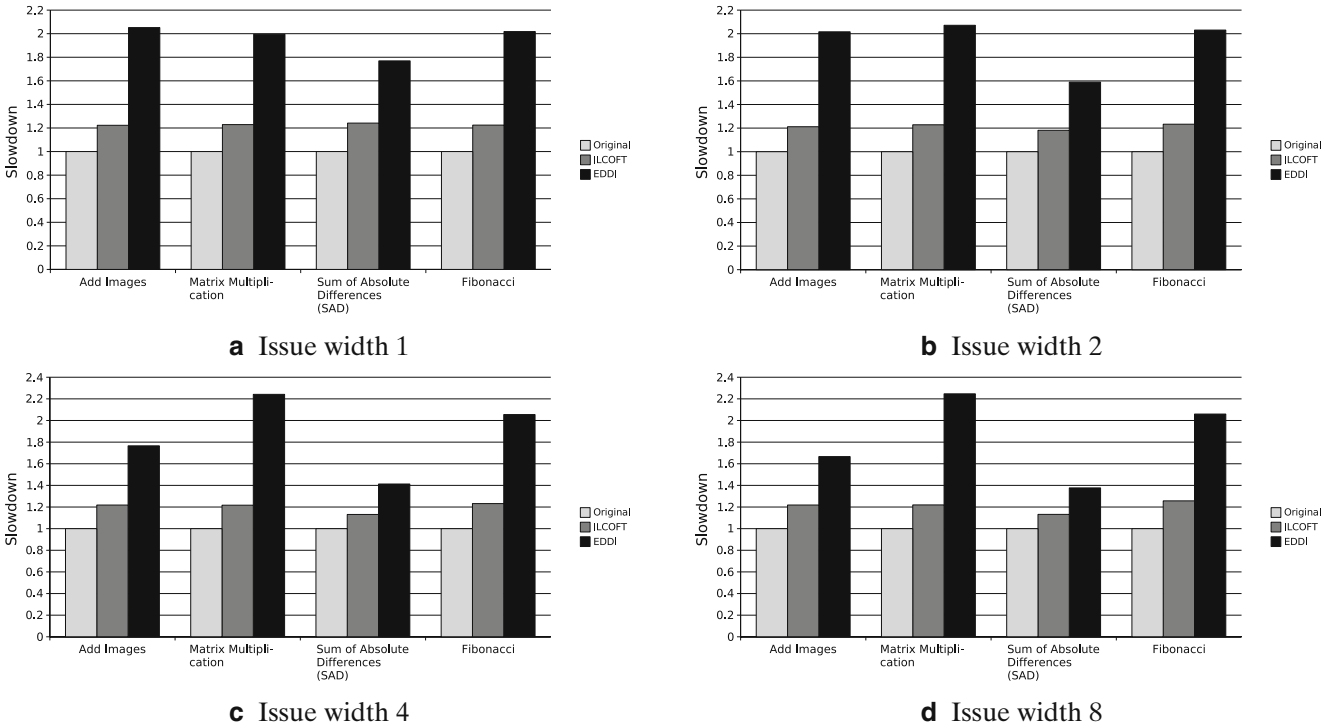


Figure 3 Slowdown of EDDI and ILCOFT-enabled EDDI versions over the non-redundant version, for varying issue widths.

EDDI) is negligible (less than 1%). The remaining contribution should be attributed to the increased number of instructions.

3.2 Energy and Power Consumption

To evaluate the energy saving of ILCOFT-enabled EDDI, we use the power analysis framework Wattch [31]. Wattch is an architectural-level micro processor power dissipation analyzer. It is a high-performance alternative to lower-level tools which are more accurate, but can only provide power estimates when the layout of a design is available. According to [31], Wattch provides a 1,000 times speedup with the accuracy within 10% of the layout-level tools. We use the default Wattch configuration. The results for the clock gating style which assumes 10% of the maximum power dissipation for unused units [31] are considered.

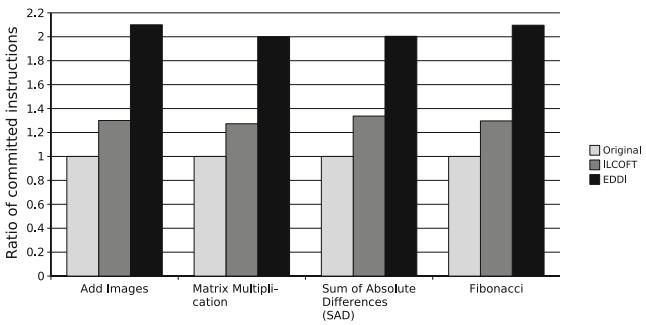
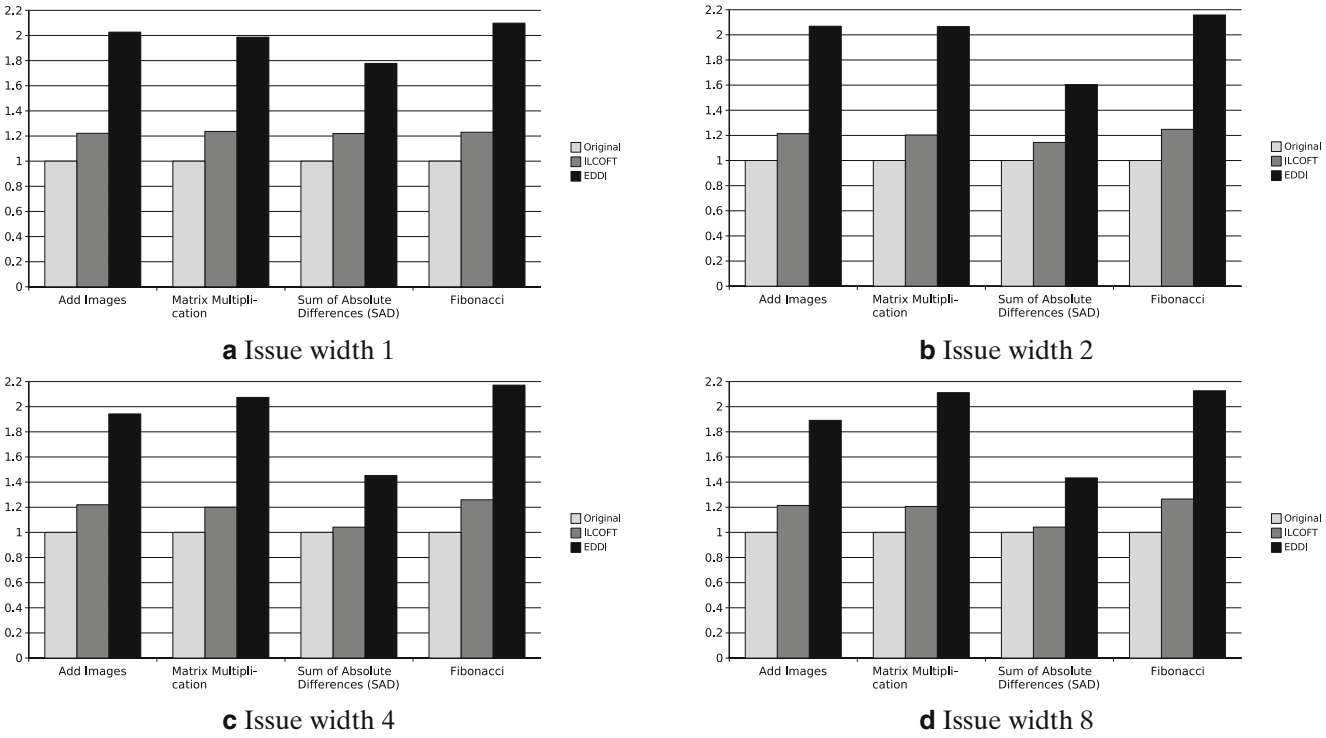


Figure 4 Committed instructions.

Energy consumption increase of EDDI and ILCOFT-enabled EDDI over the non-redundant (original) scheme is presented in Fig. 5. Four kernels, the same as in Section 3.1, are used. As expected, the energy graphs follow closely the performance graphs at Fig. 3. This is because the same factors (number of instructions and used resources) affect energy and performance. Figure 5 demonstrates that ILCOFT is able to significantly (up to 50%) reduce the energy consumption overhead.

The average power consumption per cycle of the different schemes has also been evaluated. The power consumption does not vary significantly for the three considered schemes, because the resource utilization is similar for them. The maximum fluctuation observed is 10%. As can be expected, the fluctuation is minimal with lower issue widths (no more than 3% for issue width 1), and increases with higher issue widths. This can be explained by approximately equal resource usage with lower issue widths. With higher issue widths, the resource usage varies for different schemes, due to the difference in the available ILP, and the power consumption varies accordingly. In most cases EDDI consumes more power per cycle than the other two



a Issue width 1

b Issue width 2

c Issue width 4

d Issue width 8

Figure 5 Energy consumption increase of EDDI and ILCOFT-enabled EDDI over non-redundant kernels, for varying issue widths.

schemes, because it generates more ILP, and, therefore, keeps more resources busy.

3.3 Fault Coverage Evaluation

In this section we provide an 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 by the fault injector. This is because the tested kernels have only integer arithmetic, memory and branch instructions, but 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. Fault injection into an instruction 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 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. Moreover, we simulate burst (multi-bit) faults rather than more probable single-bit faults with the purpose to represent the worst possible case.

Tables 1, 2, and 3 present the faults injection results for the three different schemes. The first column specifies the used kernels. The second 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 1,000 (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 third column shows how often faults have been

Table 1 Fault injection results for the non-redundant 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 %	Escapes % (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

detected by the FT scheme. For example, for the IA, 86.66% simulations were aborted with an error message by ILCOFT-enabled EDDI, and 100%—by EDDI. The fourth column demonstrates how often errors were detected by the simulator, for example, the application was terminated with an illegal memory access reported. The column "Undetected" contains the percentage of simulations with undetected fault(s), which shows how often the execution finished without reporting errors. The column "Application crashed" demonstrates how often an application crashed, i.e., did not produce any output. The column "Escapes" shows how often escapes occurred, i.e. an application delivered a correct result despite the presence of (undetected) fault(s). The faults have not propagated to the output. In parentheses the maximum number of undetected faults in this situation is given. The column "Max. # injected faults" gives the maximum number of faults injected per execution, before the execution finished either normally or abnormally (was interrupted reporting errors). There are usually fewer injected faults in EDDI than in other schemes, because EDDI detects and reports faults, aborting the execution, earlier. The column "Max. # undetected faults" shows the maximum number of undetected faults, which were injected but not detected; the execution is then finished without reporting errors. Most of the times these undetected faults result in corrupted application output, except the cases counted in the column "Escapes". The columns "Max. output corruption" and "Av. output corruption" present the maximum and the average output corruption caused by undetected faults. Only simulations with undetected faults, which finished without reporting errors, are considered here. This demonstrates how many undetected faults propagate to the output, and how much they affect the output. An output corruption percentage is defined as a ratio of the number of wrong output elements generated by an execution to the total number of output elements. The average output corruption is calculated as a sum of all the corruption percentages divided by the number of simulations, i.e. it is the arithmetic average. The average output corruption is used to emphasize that a very high maximum output corruption does not necessarily mean that the output is usually corrupted so much. 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 and energy consumption gain is. Among the presented kernels, the worst fault coverage (the greatest percentage of executions finished with undetected faults) appears in MM and SAD. This is because in these kernels 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 SAD used in motion estimation, a wrong motion vector leads

Table 2 Fault injection results for the ILCOFT-enabled EDDI scheme.

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (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 3 Fault injection results for the EDDI scheme.

Kernel	# sim.	Detected (FT scheme) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (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

to a wrong block, which can usually be tolerated by the user.

The exceptionally high percentage of escapes in MM (with the original and ILCOFT-enabled EDDI schemes) can be explained by the fact that most of the results (output matrix elements) are truncated when overflow occurs. Truncation masks faults by assigning the maximum possible value to any (correct or wrong) greater value. This can also be one of the reasons why MM has a relatively small percentage of detected faults with the ILCOFT-enabled EDDI scheme: 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 escapes would drop. EDDI does not have any escapes, because it detects all the faults in MM.

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 application-specific and depends solely on the algorithm employed. The IA kernel, computing every pixel value independently, without a long chain of computations, shows a very good result for ILCOFT-enabled EDDI: 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, as will be discussed later. 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. For example, in Fib, every subsequent value depends on the previous one, and thus, all the values behind the first erroneous one become wrong, independently on the FT scheme used. This leads to the extremely high final output corruption even in EDDI (see Table 3). However, only one of 581 simulations (0.17%) finished with undetected errors (2 undetected faults) with EDDI, and 7.57% of simulations—with ILCOFT-enabled EDDI, while 67.67% of unprotected executions finished with undetected errors. The single error undetected by EDDI 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. SAD 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 and unmasked errors in protected and unprotected executions affect 100% of the output. However, the unprotected execution delivered wrong result in 100% of the simulations, while EDDI-protected—only in 0.29% of the simulations. The execution protected with ILCOFT-enabled EDDI, as expected, fits in between, delivering wrong output in 44.12% cases.

To investigate the behavior under a more realistic fault appearance model, the same experiments have been conducted with a random, rather than periodic, fault injection. Faults into input or output registers were injected at random instructions, with varying fault pressure. The general impression from the results of these experiments is the same as with the periodic fault injection presented above. However, a few significant differences have been observed, which are discussed below.

For IA protected by ILCOFT-enabled EDDI, the maximum output corruption increased to 21.3%. We explain this by a larger number of faults affecting registers holding array base addresses. For MM, the maximum output corruption decreased to 30% for the non-redundant scheme, and to 50% for ILCOFT-

enabled EDDI. With the EDDI scheme, the percentage of detected faults decreased to 73.1%, and the percentage of escapes increased to 26.9%. We attribute these differences to a larger number of faults injected before truncation is performed (the effect of truncation is discussed above). For Fib, the average output corruption decreased to 41.7% for EDDI. This is because fault(s) propagated to the output in more than one simulation (1.4% simulations finished with undetected faults), affecting the output in different ways. For SAD protected by EDDI, the detected percentage dropped to 86.7%. However, most of these faults did not propagate to the output (the percentage of escapes increased to 13%).

The experimental results demonstrate that the fault coverage of ILCOFT-enabled EDDI can be significantly improved at a relatively low cost. This can be achieved by protecting the computed memory access addresses. For example, as mentioned, it could solve the corrupted output line problem in IA. The protection can be applied 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). This brings 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. Thus, 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 instruction can damage any memory location. 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 IA corrupted line problem discussed above is not solved.

4 Application-Level Validation

In this section we discuss how ILCOFT can be applied to an entire application while keeping the programming effort minimal. We estimate the advantages that it brings and evaluate the price to be paid for that.

As discussed in Section 2.2, it is infeasible that the application developer manually annotates all (block) statements with the required FT degree. Instead, everything can be automatically protected, and the programmer can focus only on some most promising application parts to minimize resource consumption at the minimum effort and reliability cost.

The running time of some applications (e.g., multimedia) is dominated by a few kernels or loops. The most time-consuming kernels often feature the natural error tolerance on which the ILCOFT idea is based. Moreover, the most time-consuming kernels are often relatively small, hence it is feasible to manage their protection manually. Thus these kernels favor ILCOFT the most, from the points of view of effectiveness, error tolerance, and minimal programming effort. A significant benefit is expected if the programmer manages manually protection of only (some of) these kernels, the rest of the application could be protected automatically.

We apply this strategy to the *cjpeg* application [32], which compresses an image file to a JPEG file. We have profiled this application and the results show that one of the most time-consuming functions on the simulated architecture is *jpeg_fdct_islow*, which implements the *inverse discrete cosine transform* (*IDCT*). This function takes from 20.7% to 23.1% of the total execution time, depending on the issue width. However, this function has a relatively small (comparing to the whole application) number of static instructions, which makes it easy to manage by hand. We apply ILCOFT-enabled EDDI only to the IDCT kernel (manually), and assume that the rest of the application is protected (automatically) by full EDDI. Further we show how this relatively small programming effort affects the whole application.

Section 4.1 analyzes the performance results, Section 4.2—energy consumption results, and Section 4.3—fault coverage results of this experiment.

4.1 Performance Evaluation

We compare the performance of the three schemes using the SimpleScalar simulator tool set [29], in the manner similar to Section 3.1. Since currently we do not have an automatic tool implementing EDDI protection, we only apply the FT schemes to the IDCT kernel in the simulation. We measure the application execution time with the IDCT kernel free of redundancy, with EDDI and ILCOFT-enabled EDDI protection. Then we use these results to derive expected results for the completely protected application. Specifically, let the total running time of cjpeg be given by

$$T_{total} = T_{idct} + T_{rest},$$

where T_{idct} is the time taken by the IDCT kernel and T_{rest} is the time of the rest of the application. Furthermore, assume that applying full EDDI (using a tool) to

the rest of the application slows it down by a factor of f, then the total running time of cjpeg protected with EDDI is given by

$$T_{total\text{-}eddi} = T_{idct\text{-}eddi} + f \cdot T_{rest},$$

where $T_{idct\text{-}eddi}$ is the measured running time of the IDCT kernel when protected with EDDI. Similarly, the total running time of cjpeg protected with ILCOFT-enabled EDDI is

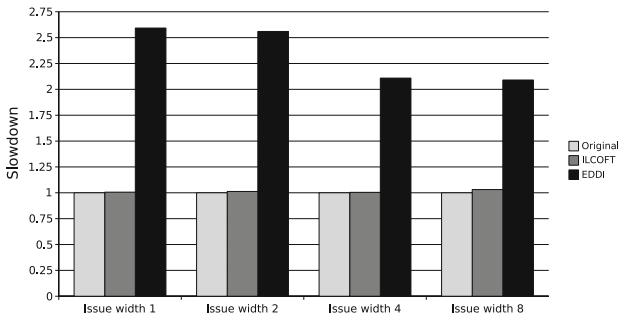
$$T_{total\text{-}ilcoft} = T_{idct\text{-}ilcoft} + f \cdot T_{rest},$$

where $T_{idct\text{-}ilcoft}$ is the measured running time of the IDCT kernel when protected with ILCOFT-enabled EDDI. In other words, for the IDCT kernel we take the measured running time and for the rest which is protected by full EDDI we assume an overhead by the factor of f (which is the same for both $T_{total\text{-}eddi}$ and $T_{total\text{-}ilcoft}$). We assume that EDDI introduces 100% overhead, which is quite pessimistic for higher issue widths, because EDDI increases the amount of available instruction-level parallelism [11]. Note that the less overhead EDDI introduces in the rest of the application, the more pronounced benefits ILCOFT-enabled EDDI brings. 100% EDDI overhead means the factor f equals 2 in our estimations.

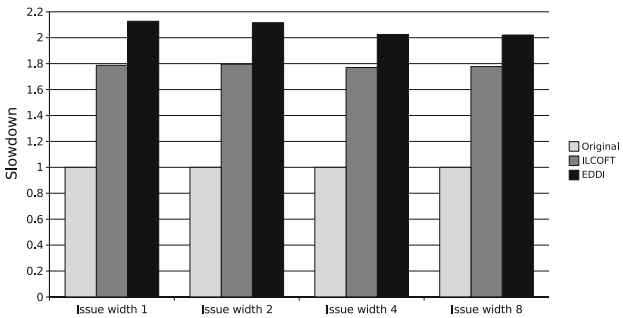
Figure 6a presents the slowdown of the IDCT kernel protected with EDDI and ILCOFT-enabled EDDI over its non-redundant version. The runtime of all the IDCT function invocations during JPEG encoding process is accumulated. This figure reflects the simulation results. Unlike in Section 3.1, this experiment uses a slightly modified version of EDDI, which does not duplicate memory. Duplicating (allocating and copying) memory for each invocation of the IDCT function would bring a significant unjustified overhead in the application, which in our opinion would not reflect the actual EDDI influence. Usually one would expect EDDI to introduce lower overhead than Fig. 6a reports, because the duplicated instructions it adds are independent of the original ones, so the available instruction-level parallelism is increased. However, for the IDCT kernel EDDI introduces overhead higher than 2 times, for example, 2.59 times for issue width 1 and 2.11 times for issue width 4 (see Fig. 6a). We attribute this to the high register utilization in IDCT. EDDI halves the number of available registers, allocating 13 of them, leading to a need in a large amount of register spilling. Besides the additional memory overhead from register spilling, the stored value of every store instruction should be checked in EDDI, which significantly increases the number of inserted branch instructions. A bad scaling with the increased issue width we attribute to the fact that the original IDCT code has enough independent instructions that can be executed in parallel, so the additional instruction-level parallelism brought by EDDI cannot be fully utilized due to the lack of computing resources.

Unlike with EDDI, IDCT is very friendly to ILCOFT (when considering only the control instructions to be critical). There are only two loops in the *jpeg_fdct_islow* function, which are not nested. Therefore ILCOFT-enabled EDDI allocates only one register (for shadow copies of the counters), duplicates only a few instructions and adds only two checks. This is very little redundancy for a function with about 350 static instructions, which leads to a negligible performance overhead over the original.

Figure 6b presents the analytically estimated performance overhead for the whole JPEG encoder. Here



a Simulated: IDCT kernel slowdown, accumulated for all its invocations in JPEG encoding.



b Estimated: JPEG encoder slowdown with protection applied to the whole application. Based on the assumption that for the rest of the application (except the IDCT kernel), EDDI introduces 100% overhead.

Figure 6 Slowdown of EDDI and ILCOFT-enabled EDDI versions over the non-redundant version, for varying issue widths.

Table 4 Fault injection results for the JPEG encoding.

	# sim.	Detected (FT scheme) %	Detected (application) %	Detected (simulator) %	Undetected %	Application crashed %	Escapes % (max. # faults)	Max. # injected faults	Max. # undetected faults
ILCOFT	100	0	59	4	35	0	9(4)	9	8
EDDI	100	98	0	27	0	0	0	147	0

the IDCT kernel is protected with either EDDI or ILCOFT-enabled EDDI, and the rest of the application—with EDDI. It shows that applying ILCOFT to only the IDCT kernel of EDDI-protected JPEG encoder is able to deliver a performance gain of 14% on average.

4.2 Energy Consumption

Similar to Section 3.2, we obtained the energy consumption results with Wattch [31]. The results show a behavior similar to that of the performance in Fig. 6. On average, the JPEG encoder with the IDCT kernel protected by ILCOFT-enabled EDDI consumes about 14% less energy than with EDDI.

4.3 Fault Coverage Evaluation

We perform experiments similar to those in Section 3.3, injecting faults regularly (from once per 500 thousand to once per 50 million instructions) into the input and output registers of the instructions within the IDCT kernel. The fault frequency decreases with each simulation. Table 4 presents the fault injection results for 100 simulations. The presentation is similar to that of Table 1.

Table 4 shows that EDDI detected almost all the faults (98%). The effects of the other 2% faults have been masked, did not propagate to the output, or were detected by the simulator before the FT scheme. ILCOFT-enabled EDDI did not detect any faults in our simulation. We explain this by the fact that the number of checks performed within the kernel is very low as compared to EDDI.

The column "Detected (application)" shows that 59% of the faults in ILCOFT-enabled EDDI have been detected by the application, reporting an out-of-range DCT coefficient. Nothing has been detected by the application in EDDI, because the faults have been caught earlier by EDDI or the operating system. The column "Detected (simulator)" demonstrates that for both schemes some faults have been detected by the simulator (operating system), reporting, for example, an illegal memory access. From the column "Application crashed" it can be seen that the application never crashed due to the faults, neither for EDDI nor for ILCOFT-enabled EDDI.

It may appear surprising that in the column "Max. # injected faults", the maximum number of faults injected per simulation is much larger for EDDI than for ILCOFT-enabled EDDI. This is due to the way our error handler works: when EDDI detects a fault, it reports an error and returns from the running IDCT function, but does not stop the whole application. In this way we are able to see if EDDI detects faults in future IDCT invocations. However, unlike in EDDI, undetected faults in ILCOFT-enabled EDDI easily propagate to the points where the wrong values are used in loads and stores, which triggers a simulator (operating system) exception, and the application stops. Thus, the simulation is shorter, and the fault injector is not able to inject more faults.

Thirty-five percent of the ILCOFT scheme simulations ended with undetected faults, which were either masked or propagated to the output. Figure 7 depicts one of the most corrupted output JPEG images produced by our simulations. It was produced with fault frequency of once per 10 million instructions, and this particular simulation produced the maximum

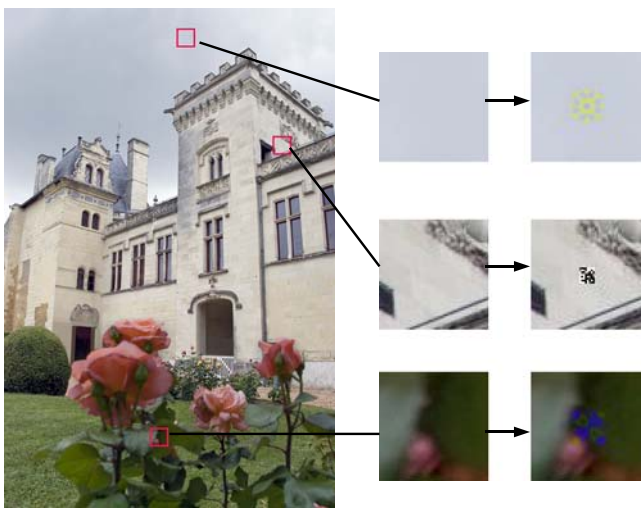


Figure 7 Output corruption due to the undetected faults in IDCT.

number of undetected faults (equal to 8). The three areas where we were able to visually recognize corruption are marked with squares. On the right, the magnified versions of these areas are shown, in the original image (left column) and the corrupted image (right column).

We believe that the possibility to end up with this kind of output corruption is quite an acceptable price for a significant speedup and energy saving which ILCOFT applied to the protection of the IDCT kernel provides. This is under the assumption of relatively low requirements to the output image quality, which can be quite appropriate in embedded systems, PCs and other systems not designed for critical missions. For applications with very high requirements to the output image quality JPEG is not a good choice anyway, because it is originally lossy. The other expectation is a relatively low fault rate which is anticipated in the foreseeable future in normal environments (ILCOFT does not target extreme cases such as environments with a high radiation). The low fault rate means that most of the time redundancy is not useful, but only brings overhead. In this situation, reducing time and energy overhead, still being guaranteed against severe crashes, but increasing the chance of tolerable errors, makes sense.

5 Conclusions

In this work we have proposed an instruction-level, rather than application-level, configurability of FT techniques. 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 (or left unprotected) to improve the application performance and save resources. 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 could be adapted to support ILCOFT. We also proposed a way how a programmer could specify the desired degree of FT in a high-level language or assembly code, and indicated how a compiler could apply FT techniques to control code automatically.

The experimental results have demonstrated that ILCOFT is able to significantly improve an application performance and reduce the energy consumption when applying a higher FT degree to its critical parts (instructions) only. At the kernel level, the performance and energy dissipation improved up to 50%, and at the application level—up to 16%. In the application-level experiments, this improvement is achieved by applying ILCOFT to only one of the most time-consuming kernels, minimizing the programmer effort. This indicates that adaptation of only one kernel provides a significant application-level improvement.

The price to be paid for the performance and energy gains provided by ILCOFT is the decreased fault coverage. The experimental results have 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. In some cases the output corruption introduced by ILCOFT is tolerable, in others it is not acceptable. Finally, we have demonstrated that adding memory access address protection in ILCOFT-enabled EDDI could 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.

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

References

1. Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D., & Alvisi, L. (2002). 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* (pp. 389–398). Washington, DC, USA.
2. Rao, T. R. N., & Fujiwara, E. (1989). *Error-control coding for computer systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.
3. Hareland, S., Maiz, J., Alavi, M., Mistry, K., Walsta, S., & Dai, C. (2001). Impact of CMOS process scaling and SOI

on the soft error rates of logic processes. *VLSI technology. Digest of technical papers* (pp. 73–74).
4. Saxena, N. R., & McCluskey, E. J. (1998). Dependable adaptive computing systems—the ROAR project. In *Proc. IEEE systems, man, and cybernetics conf* (vol. 3, pp. 2172–2177) (October).
5. Sundaramoorthy, K., Purser, Z., & Rotenberg, E. (2000). Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices, 35*(11), 257–268.
6. Purser, Z., Sundaramoorthy, K., & Rotenberg, E. (2000). A study of slipstream processors. In *MICRO-33: Proc. 33rd annual ACM/IEEE int. symp. on microarchitecture* (pp. 269–280). New York, NY, USA.
7. Breuer, M. A., Gupta, S. K., & Mak, T. M. (2004). Defect and error tolerance in the presence of massive numbers of defects. *IEEE Design and Test of Computers, 21*(3), 216–227.
8. Chung, H., & Ortega, A. (2005). Analysis and testing for error tolerant motion estimation. In *DFT-05: Proc. 20th IEEE Int. symp. on defect and fault tolerance in VLSI systems* (pp. 514–522). Washington, DC, USA (October).
9. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., & August, D. I. (2005). SWIFT: Software implemented fault tolerance. In *CGO '05: Proc. of the int. symp. on code generation and optimization* (pp. 243–254). Washington, DC, USA.
10. Oh, N., & McCluskey, E. J. (2002). Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability, 51*(4), 392–402 (December).
11. Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability, 51*(1), 63–75 (March).
12. Lu, D. J. (1982). Watchdog processors and structural integrity checking. *IEEE Transactions on Computers, C-31*(7), 681–685 (July).
13. Mahmood, A., & McCluskey, E. J. (1988). Concurrent error detection using watchdog processors–a survey. *IEEE Transactions on Computers, 37*(2), 160–174 (February).
14. von Neumann, J. (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata studies*, volume 34 of *annals of mathematics studies*, (pp. 43–98). Princeton, NJ: Princeton University Press.
15. Johnson, B. W. (1989). *Design and analysis of fault-tolerant digital systems*. Addison-Wesley (January).
16. Hennessy, J. L., & Patterson, D. A. (2003). *Computer architecture, a quantitative approach* (3rd ed). Morgan Kaufmann (May).
17. Franklin, M. (1995). A study of time redundant fault tolerance techniques for superscalar processors. *IEEE Int. workshop on defect and fault tolerance in VLSI systems* (pp. 207–215) (November).
18. Ray, J., Hoe, J. C., & Falsafi, B. (2001). Dual use of superscalar datapath for transient-fault detection and recovery. *MICRO-34* (pp. 214–224) (December).
19. Austin, T. M. (1999). DIVA: A reliable substrate for deep submicron microarchitecture design. In *MICRO-32: Proc. 32nd annual ACM/IEEE Int. symp. on microarchitecture* (pp. 196–207). Washington, DC, USA (June).
20. Chatterjee, S., Weaver, C., & Austin, T. (2000). Efficient checker processor design. In *MICRO-33: Proc. 33rd annual ACM/IEEE int. symp. on microarchitecture* (pp. 87–97). New York, NY, USA.
21. Weaver, C., & Austin, T. (2001). A fault tolerant approach to microprocessor design. *Dependable systems and networks* (pp. 411–420) (July).
22. Tullsen, D. M., Eggers, S. J., & Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-95: Proc. 22nd annual int. symp. on computer architecture* (pp. 392–403). New York, NY, USA.
23. Rotenberg, E. (1999). AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTCS-29* (pp. 84–91). Madison, Wisconsin, USA (June).
24. Reinhardt, S. K., & Mukherjee, S. S. (2000). Transient fault detection via simultaneous multithreading. In *ISCA-00: Proc. 27th annual int. symp. on computer architecture* (pp. 25–36). New York, NY, USA.
25. Vijaykumar, T. N., Pomeranz, I., & Cheng, K. (2002). Transient-fault recovery using simultaneous multithreading. In *ISCA-02: Proc. 29th annual int. symp. on computer architecture* (pp. 87–98). Washington, DC, USA.
26. Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., & Chang, K. (1996). The case for a single-chip multiprocessor. In *ASPLOS-VII: Proc. seventh int. conf. on architectural support for programming languages and operating systems* (pp. 2–11). New York, NY, USA.
27. Reynolds & Metze, G. (1978). Fault detection capabilities of alternating logic. *IEEE Transactions on Computers, C-27* (12), 1093–1098 (December).
28. Patel, J. H., & Fung, L. Y. (1982). Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers, C-31*(7), 589–595 (July).
29. Burger, D., & Austin, T. M. (1997). The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News, 25*(3), 13–25.
30. Austin, T., Larson, E., & Ernst, D. (2002). SimpleScalar: An infrastructure for computer system modeling. *Computer, 35*(2), 59–67.
31. Brooks, D., Tiwari, V., & Martonosi, M. (2000). Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-00: Proc. of the 27th annual int. symp. on computer architecture* (pp. 83–94). New York, NY, USA.
32. Independent JPEG Group webpage. http://www.ijg.org/.

Demid Borodin was born in Samarkand, USSR, in 1982. He received the M.Sc. degree in Computer Engineering from Delft University of Technology, Delft, The Netherlands, in 2005. Currently he is a Ph.D. student in Computer Engineering Laboratory of the Faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology, The Netherlands. His current research focuses on fault tolerance of computing systems. The interests also include application-specific instruction set architectures, parallel architectures, and computer architecture in general.

B. H. H. (Ben) Juurlink is an associate professor in the Computer Engineering Laboratory of the Faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology, The Netherlands. He received the M.Sc. degree in Computer Science, from Utrecht University, Utrecht, The Netherlands, in 1992, and the Ph.D. degree also in Computer Science from Leiden University, Leiden, The Netherlands, in 1997. His research interests include instruction-level parallel processors, application-specific ISA extensions, low power techniques, and hierarchical memory systems. He has (co-)authored more than 50 papers in international conferences and journals and is a senior member of the IEEE, a member of the ACM, and a member of the HiPEAC Network of Excellence.

Said Hamdioui received the M.S.E.E. and Ph.D. degrees (both with honors) from the Delft University of Technology, Delft, The Netherlands. He is currently with the Delft University of Technology. He has more than seven years with industry before attending Delft University. He worked for Intel CA, Philips Semiconductors France and for Philips Nijmgen in The Netherlands. Dr. Hamdioui published one book and over 50 technical papers. His research interests include VLSI test and reliability, deep-submicron CMOS IC design and test, fault /defect tolerance, nano devices design and test. Dr. Hamdioui is a member of the IEEE and the IEE, and was the recipient of the European Design Automation Association (EDAA) Outstanding Dissertation Award for 2001.

Stamatis Vassiliadis was born in Manolates, Samos, Greece, in 1951. He was a Chair Professor in the Faculty of EEMCS at Delft University of Technology, The Netherlands. He previously served in the Electrical Engineering faculties of Cornell University, Ithaca, NY, and the State University of New York (SUNY), Binghamton. For a decade, he worked with IBM, where he was involved in a number of advanced research and development projects. Dr. Vassiliadis has received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engineering/scientific hardware design. His 72 U.S. patents rank him as the top all-time IBM inventor. He passed away in April 2007.