Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor *

Demid Borodin
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Tel: +31 152787362, Fax: +31 152784898.
D.Borodin@TUDelft.nl

B.H.H. (Ben) Juurlink†
Embedded Systems Architectures
Institute for Computer Engineering and Micro-Electronics
Technische Universität Berlin
Franklinstraße 28/29, D-10587 Berlin, Germany
Tel: +49 3031425538, Fax: +49 3031422943
Juurlink@CS.TU-Berlin.de

ABSTRACT

Due to modern technology trends, fault tolerance (FT) is acquiring an ever increasing research attention. To reduce the overhead introduced by the FT features, several techniques have been proposed. One of these techniques is Instruction-Level Fault Tolerance Configurability (ILCOFT). ILCOFT enables application developers to protect different instructions at varying degrees, devoting more resources to protect the most critical instructions, and saving resources by weakening protection of other instructions. It is, however, not trivial to assign a proper protection level for every instruction. This work introduces the notion of *Instruction Vulnerability Factor* (*IVF*), which evaluates how faults in every instruction affect the final application output. The IVF is computed off-line, and is then used by ILCOFT-enabled systems to assign the appropriate protection level to every instruction. IVF releases the programmer from the need to assign the necessary protection level to every instruction by hand. Experimental results demonstrate that IVF-based ILCOFT reduces the instruction duplication performance penalty by up to 77%, while the maximum output damage due to undetected faults does not exceed 0.6% of the total application output.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance; D.2.4 [**Software/ Program Verification**]: Reliability

*This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6), and by the Dutch Ministry of Economic Affairs in the context of the 3DIM^3 CATRENE project #CT105.

†The work was partially performed while the second author was affiliated with Delft University of Technology.

CF'10, May 17–19, 2010, Bertinoro, Italy.

General Terms

Design, Performance, Reliability, Verification

Keywords

performance, redundancy, fault detection, instruction vulnerability, selective protection

1. INTRODUCTION

Fault tolerance (FT) of computing systems is receiving an instantly increasing attention since recently [1]. This is due to current technology trends such as increasing the chip density by shrinking the feature size and power supply minimization. When the transistor feature size decreases, they become vulnerable to multiple external disturbances and internal problems. For example, radiation which could not affect larger transistors and wires is able to cause faults in smaller ones, internal crosstalk becomes more common, etc. Minimization of the power supply increases the chance of bit flips. As a result, FT is becoming increasingly important. While strong FT techniques used to be applied only to special-purpose high-end computing systems, and only a few techniques such as Error Correcting Codes (ECC) [2] in memories were relatively widely used, they are becoming common nowadays.

FT is often based on some form of redundancy, which means that it introduces a certain hardware overhead, performance penalty and/or energy consumption increase. While acceptable for expensive critical computing systems, it should be avoided as much as possible in commodity systems. Therefore, FT overhead minimization is receiving a lot of attention. Many techniques, sometimes trading FT for overhead minimization, have been proposed. One of these techniques, Instruction-Level Fault Tolerance Configurability (ILCOFT) [3, 4], requires the application developer to assign the required protection level to every assembly instruction, or every high-level programming language statement, or set of instructions/statements, and protects them accordingly. This is useful for naturally error tolerant applications. For example, in multimedia applications, much of the computations do not strictly require absolute correctness. Many faults can lead to slight output imperfections that in some cases are not even noticeable for a human. For example, if a few pixels in a large image are wrong, they are likely to be unnoticed. By enforcing the protection of more critical instructions and/or weakening the protection of less critical ones, ILCOFT achieves a higher overall reliability and/or minimizes the protection overhead. Depending on the FT method used, this can be performance, energy, and/or area overhead.

A disadvantage of the ILCOFT scheme as presented in [3, 4] is the manual protection level assignment required from the application developer. To process (a large part of) a relatively large application, significant effort is required. In addition, this process is error-prone because it entirely relies on the programmer's judgment of every instruction's vulnerability. An alternative automatic method is also proposed in [3, 4] in which the compiler assigns the required protection level. Because the compiler cannot evaluate the criticality of data processing instructions, it is assumed that only control flow instructions need strong protection. Thus, all control flow instructions (branches, jumps, function calls, etc.) and instructions on which they depend (address calculations, condition evaluations, etc.) are assigned a high protection level. Sundaram et al. [5] discuss such compiler analysis in detail. This automatic compiler-based method, however, is based on the assumption that no data manipulating instructions are critical, which is not safe. In many applications faults in data processing instructions can corrupt the whole application output, in which case, even though the application does not crash, it is not usable. This approach can only be safely applied to a very limited set of applications. A more sophisticated method to assess every instruction's criticality is required for most applications.

To address the instruction criticality assessment problem, this work introduces the notion of *Instruction Vulnerability Factor* (*IVF*). IVF is analogous to the Architecture Vulnerability Factor (AVF) [6], but addresses application instructions instead of hardware structures. AVF estimates the probability that a fault in a particular hardware structure will result in an error visible in the final application output. Experiments demonstrate that different processor structures have different AVFs [6]. For example, unlike faults in ALUs, faults in branch predictors can only result in a slightly reduced performance, but cannot damage the final application output. We suggest that a similar metric can be applied to the application's instructions. Faulty results of different instructions affect the final application output in different ways. IVF measures how much of the final output is corrupted due to faults in every instruction.

ILCOFT can be applied to different instructions based on their IVF. Instructions with a higher IVF should be protected better, while others can be assigned a weaker protection level to reduce the overhead.

To compute the IVF for every instruction, off-line profiling is performed. It can be done either in a simulation environment or in real hardware capable of injecting faults. Fault(s) should be injected into every instruction, one per experiment, and the resulting application output be used to estimate the instruction IVF. Multiple experiments per instruction are preferable to obtain statistically valid results. To reduce the amount of work this process requires, only the most time consuming application parts can be processed. As shown in Section 3, this is able to provide significant application-level advantage. This procedure has to be performed off-line, only once per application. The result (IVF value for all the considered instructions) is then stored and distributed together with the application binary code. At runtime, every instruction is protected at the level required by its IVF value.

The remainder of this paper is structured as follows. Section 2 describes the IVF estimation performed in this work, and how it is then used to reduce the overhead of the instruction duplication error detection technique. Section 3 presents the experimental results evaluating the performance penalty reduction and fault coverage of the proposed IVF-based ILCOFT technique. Finally, Section 4 draws conclusions and describes future work.

2. IVF AND IVF-BASED ILCOFT

This section introduces the IVF and demonstrates how it can be used as a base for ILCOFT. Section 2.1 discusses how the IVF can be estimated. Section 2.2 presents the instruction duplication error detection scheme, which is adapted to support IVF-based ILCOFT in Section 2.3.

2.1 IVF Estimation

IVF estimation requires monitoring how different faults in every instruction propagate to the final application output. In other words, how much of the output is corrupted due to faults in every instruction. This can be achieved using fault injection experiments, either in a simulation environment or on fault injection-enabled hardware. Alternatively, this can also be done in software without simulations. To simulate faults in software, the correct machine instructions can be substituted with other instructions producing wrong results in the same output registers. The simulator-free software solution can be expected to significantly speed up the IVF estimation process, but it reduces the fault injection flexibility (does not provide access to simulated hardware structures where faults happen).

Ideally, all the possible faults have to be injected one at a time into every executed instruction, and the corresponding output corruption measured. Moreover, some faults in dependent instructions can affect each other, thus combinations of faults in different instructions should also ideally be examined. This, however, would require an enormous number of experiments which is not feasible. Experimental results (see Section 3) demonstrate that injecting a random fault into every instruction provides a sufficiently accurate IVF estimation. It is desirable to perform multiple experiments with every instruction to achieve statistically valid results, because different faults in the same instruction can affect the execution in different ways.

In this work, the IVF estimation is performed on the *sim-outorder* simulator from the SimpleScalar tool set [7]. During every simulation, the output of one instruction execution is assigned a random value, simulating the worst case (a multi-bit) transient fault. Faults are only injected into instructions producing results, such as arithmetic operations and memory loads. Memory stores and jumps are not affected, because the error detection scheme used in this work does not cover faults in these instructions. The final application output is then compared to the correct one, and the output corruption is measured. Depending on the application, this can be the percentage of corrupted bytes in the output (in an image for instance), or the percentage of corrupted output items, such as matrix elements. The output corruption percentage is then saved as the instruction's IVF. When multiple experiments per instruction are performed, the average IVF is saved for all the experiments.

The average IVF value is computed for every static instruction within the considered code segment. This requires a large number of simulations, especially if multiple experiments per instruction are conducted (as is desirable). This, however, has to be done only once per application, and can be done off-line. The collected statistics are then saved together with the application binary code and used at runtime. For large applications, it is not even necessary to compute the IVF of every instruction. It can be estimated only for the instructions within the most time consuming application parts, such as multimedia kernels. Section 3.3 demonstrates that this approach achieves significant application-level performance improvements.

The IVF information can be saved in a program binary code as a separate table. This table maps application instructions (identified by their PC) to the corresponding IVF values, or to the required protection levels. Storing the protection levels is likely to take less

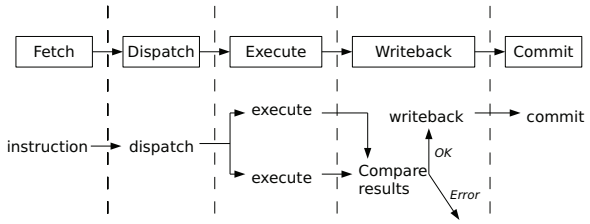

Figure 1: Instruction duplication.

space (for example, only one bit is needed if two protection levels are available), and no logic is needed to determine the protection level from the IVF value. On the other hand, storing the IVF value provides more flexibility, in that the system can be configured at runtime to define the threshold IVF values controlling the required protection levels. The IVF information stored in a separate table needs to be loaded into a special hardware buffer (possibly simultaneously assigning the protection levels and storing them instead of IVF values to reduce the storage requirements). Then, the table must be looked up to determine the proper protection level for every executed instruction. The table can be compressed. For example, with only two protection levels available in the system (of which one is default), only PCs of instructions with the non-default required protection level have to be stored. Alternatively, instead of keeping the IVF data in a separate table, it might be preferable to change the instruction format, to add the required protection levels there.

2.2 Instruction Duplication

In this work a hardware ILCOFT scheme is used. Instruction duplication in the pipeline is used as a time-redundant error detection technique, and its overhead is minimized using the IVF information collected as described in Section 2.1. The instruction duplication scheme is based on [8], but executes every instruction twice rather than duplicating them in the dynamic scheduler. This requires less space in the dynamic scheduler, but does not protect against faults in it. Figure 1 depicts the instruction execution steps with the corresponding activities performed by every instruction. A fetched instruction proceeds normally until the execution stage. There the instruction is kept in the RUU until it has been issued twice. When the results of both executions are available, they are compared in the Writeback stage, and the instruction commits if no errors are detected.

What happens if the results do not match depends on the goals of the system. A system targeting fail-safe operation would signal an error and halt. If FT is required, another copy of the questioned instruction could be created and executed, performing a majority voting on all the obtained results according to the Triple Modular Redundancy (TMR) scheme [9, 10], or a different form of recovery could be initiated. This, however, is outside the scope of this work.

2.3 IVF-Based Selective Instruction Duplication

To support ILCOFT, the FT technique(s) used in the system have to be able to apply different protection levels to different instructions. The higher the IVF is, the stronger the protection technique applied to the instruction should be. In the simplest case, only two protection levels are available: an instruction is either protected or left unprotected. This case is very practical, because more complex protection schemes with multiple protection levels can be very

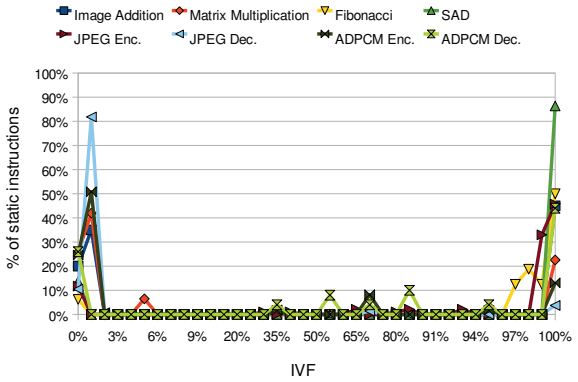

Figure 2: Histogram of IVF values for different applications and kernels.

expensive. In this work such simple protection scheme based on instruction duplication (see Section 2.2) is used, called *IVF-Based Selective Instruction Duplication* (*IVF-SID*). Depending on its IVF, an executed instruction is either duplicated and the result is verified, or executed only once without error detection.

Note that two terms are used in this work: IVF-based ILCOFT and IVF-SID. IVF-based ILCOFT is a general ILCOFT technique which decides how critical instructions are based on their IVF values. IVF-based ILCOFT can be applied to many different FT techniques able to protect individual instructions. IVF-based ILCOFT applied to the instruction duplication error detection technique is called IVF-SID.

The simple protected/unprotected scheme requires to establish a threshold IVF value, further referred to as IVF_{thr}. Instructions with IVF above IVF_{thr} should be protected, and instructions with IVF below IVF_{thr} are left unprotected. The IVF_{thr} value should be carefully chosen to guarantee that the amount of application output corruption it represents is indeed tolerable. Ideally it should be determined specifically for every application, because different applications have varying output damage tolerance.

Figure 2 shows the histogram of IVF values for different kernels (the kernels are described in Section 3.1). It clearly illustrates that the majority of IVF values are at the extremes, they are either larger than 99% or less than 1%. Some kernels (Image Addition and SAD) do not have any instructions with IVF values in between. Most instructions either corrupt (almost) the whole output or less than 1% of it. This indicates that 1% is a good IVF_{thr} value. Damage of 1% of the output can be considered tolerable for many application domains, including many multimedia applications. Increasing the IVF_{thr} to, for example, 10% would not lead to significant changes, because most kernels do not have any instructions with an IVF in the range of 1% to 10% (see Figure 2). Thus, an IVF_{thr} value of 1% is used for IVF-SID in this work. Instructions whose wrong results corrupt less than 1% of the application output are not protected.

The IVF values distribution shown in Figure 2 suggests that a simple protected/unprotected ILCOFT scheme is sufficient. Multiple available protection levels would not be very useful for most of the benchmarks shown in Figure 2, because they have only a few instructions with medium IVF values. On average 44% of the instructions have an IVF below 1%, and 39% of the instructions have an IVF above 99%. Most of the other instructions have an IVF below 5% or above 95%. Only the ADPCM encoder and decoder have a significant number of instructions (10% or more) with

Table 1: Processor configuration.

Fetch/Dec./Issue Width	2, 4, or 8
# of Int. ALUs	2, 4, or 8
# of Int. Mult./Div.	1
# of FP ALUs	1
# of FP Mult./Div.	1
RUU Size	64
Memory Latency	112 cycles (first chunk), 2 cycles (subsequent chunks)
L1 Data Cache	32 KB, 2-way set associative
L1 Instruction Cache	32 KB, 2-way set associative
L2 Unified Cache	512 KB, 4-way set associative

an IVF in the range 40-90%. For such applications, the following scheme could be used: no or minimum protection for instructions with an IVF below 1%, average protection for instructions with an IVF between 1% and 99%, and maximum protection for instructions with an IVF above 99%.

3. EXPERIMENTAL EVALUATION

This section experimentally evaluates the proposed IVF-SID method. Section 3.1 describes the used simulator and benchmarks. Section 3.2 explains the details of the employed IVF estimation method. Section 3.3 demonstrates the performance improvements IVF-SID achieves compared to duplicating all instructions. Section 3.4 evaluates its fault coverage. Finally, Section 3.5 compares IVF-SID with a manual ILCOFT method.

3.1 Experimental Setup

Simulations are performed using the *sim-outorder* simulator from the SimpleScalar tool set [7]. The processor configuration (not very powerful because small kernels are considered) is shown in Table 1.

Four kernels and four applications are used as benchmarks. Image Addition, Matrix Multiplication (with rather small input matrices of the size 20×10 and 10×20 to reduce the simulation time), and Sum of Absolute Differences (SAD) are kernels very often used in multimedia applications. The fourth kernel computes the Fibonacci numbers, which are widely used in science, and even in financial market trading and music [11]. Encoders and decoders for JPEG image compression and ADPCM sound compression are the full applications used, taken from the MediaBench benchmark suite [12].

The kernels are chosen to represent a range of algorithms: from (very) tolerable to faults to hardly tolerable to faults. Image Addition contains a large number of independent calculations, which makes it tolerant to faults: a fault in most of the computational instructions can only affect a single output element. Matrix Multiplication has many independent as well as many dependent operations. The output elements are, however, computed independently, thus a fault in one of them does not affect the others. Every Fibonacci number depends on all the previously computed numbers, and thus, any fault leads to the corruption of all subsequent values. SAD outputs a single value which depends on the whole computation sequence, and is either correct or wrong. Any fault is SAD leads to a wrong result, thus, SAD is the most vulnerable kernel.

3.2 IVF Calculation

The IVF is estimated as described in Section 2.1. For the full applications, to reduce the simulation time and demonstrate that it still provides useful results, IVF-based ILCOFT is applied only to the most time consuming functions. The applications are profiled and the most demanding functions identified (for example, the *forward_DCT* function in the JPEG encoder, and the *jpeg_idct_islow* function in the JPEG decoder). The IVF values have been estimated and are later used by ILCOFT only for the instructions within these functions. In other parts of the applications all instructions have been duplicated as described in Section 2.2.

To calculate the IVF of every instruction, ten fault injection experiments per static instruction have been conducted for the kernels, and three experiments per instruction for full applications. To evaluate the accuracy of this method, an additional experiment is performed with one of the kernels (Matrix Multiplication). This experiment attempts to approach the ideal (exhaustive) case taking into account all possible faults and even combinations of faults in different instructions (see Section 2.1). First, for every evaluated static instruction, one thousand experiments injecting a single random fault into it are executed. Then, faults are injected in two different instructions at a time. This is achieved in the following way. Each evaluated instruction is coupled with every other instruction in the kernel whose originally estimated IVF (from the first experiment with ten injections per instruction) does not exceed its own originally estimated IVF. The obtained output corruption contributes to the IVF of the evaluated instruction. Instructions with greater originally estimated IVF are not coupled, because they affect the output more than the evaluated instruction. Fifty iterations of this double-injection experiment are performed, resulting in 350 to 1500 simulations per instruction.

The results of the short Matrix Multiplication IVF estimation experiment closely follow the results of the large experiment. The average difference between the obtained IVF values is 0.5%. The maximum difference is 7% (it is so large only for two instructions). Hence, the short experiment performs with a sufficient accuracy. Fault injection experiments in Section 3.4 confirm that a high level of reliability is achieved using the short IVF estimation experiments.

To provide an insight on the time the IVF estimation process takes, the fault injection experiment simulation time has been measured for one randomly chosen kernel and one application. For the Fibonacci kernel, an experiment injecting one fault into one instruction took 0.8 seconds, and for the JPEG Encoding application it took 136.7 seconds. Note that this is for the case when simulation produces a complete output (does not crash due to faults). Simulations crashing (or producing incomplete outputs) due to faults take less time. On the other hand, there are also faults substantially increasing the simulation time. For example, a fault that significantly increases the value of a variable controlling the exit condition of a large loop can have such effect.

3.3 Performance

Instruction duplication (see Section 2.2) requires that every dynamic instruction is executed twice. Since duplicated instructions are independent, they can be executed in parallel, if sufficient computational resources are available. Thus, instruction duplication increases the amount of instruction-level parallelism (ILP) available in the application. Unless the application originally has a very limited amount of ILP, instruction duplication is likely to introduce a significant performance penalty due to the lack of computational resources available to execute the instruction duplicates. By avoiding the re-execution of instructions with an IVF smaller than 1%, IVF-SID reduces the overhead of instruction duplication.

Figure 3 shows the performance overhead of full instruction duplication and IVF-SID over the original execution without any protective redundancy. The benchmarks are executed on a system with four integer ALUs, and a fetch/decode/issue width of four. Fig-

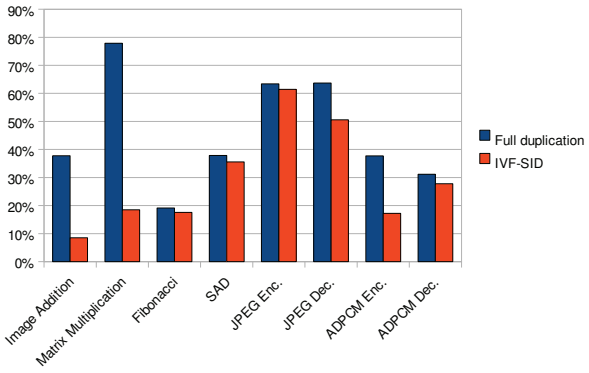

Figure 3: Performance penalty of instruction duplication and IVF-SID compared to redundancy-free execution. System with 4 integer ALUs, and fetch/decode/issue width of 4.

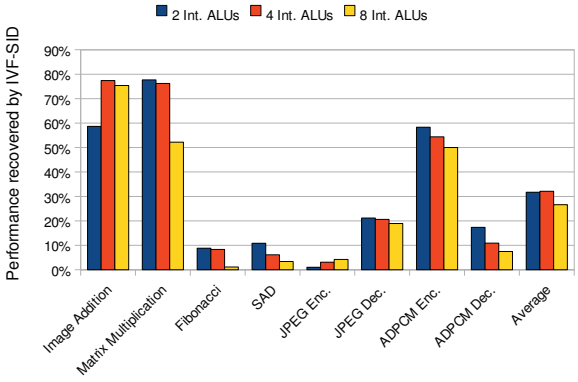

Figure 4: Performance penalty reduction achieved by IVF-SID over instruction duplication on different processor configurations. Fetch/decode/issue width matches the number of integer ALUs.

ure 3 demonstrates that for all the benchmarks, IVF-SID recovers a certain amount of the performance overhead due to instruction duplication. For example, if the full duplication is 80% slower than the redundancy free execution, and IVF-SID is 40% slower, than the amount of performance it recovers is 50%. Figure 4 shows the amount of recovered performance for different benchmarks. Three machine configurations are considered: with two, four and eight integer ALUs. To balance the machine organization, the fetch, decode, and issue width matches the number of integer ALUs (see Table 1).

For the kernels, the recovered performance varies from 1.1% (for Fibonacci, on a system with eight integer ALUs) to 77.7% (for Matrix Multiplication, on a system with two integer ALUs). Full applications recover from 1% (JPEG Encoder, system with two integer ALUs) to 58.3% (ADPCM Encoder, system with two integer ALUs).

The amount of recovered performance depends on the IVF distribution of the application, and the amount of ILP available in the redundancy-free application. The more instructions with low IVF are found in the application, the fewer executed instructions are duplicated, and more performance is recovered. The distribution of instructions with different IVF values in the benchmarks is shown in Figure 2.

In SAD, almost 86.4% of the static instructions have an IVF of above 99% (and thus definitely need to be duplicated in IVF-SID), and only 13.6% of the instructions have an IVF lower than 1% (and thus do not need to be duplicated). This is because SAD produces only one output, which can be either correct (100% of the output is correct) or wrong (100% of the output is damaged). Any fault which propagates to the final SAD output damages 100% of it. Therefore, only faults that do not propagate to the final output at all (damage unused data, or faults that are masked) can be tolerated in SAD. We call these faults *escapes*. Due to these characteristics, SAD is one of the worst performing benchmarks: only 3.4% to 10.8% of performance is recovered in IVF-SID.

A similar situation appears with the Fibonacci numbers generator, for which IVF-SID recovers from 1.1% to 8.8%. The generator produces a series of numbers, every one of which depends on the previous values. Thus, an error appearing in the beginning of the sequence damages all the subsequent numbers. Only 6.3% of the static instructions in the Fibonacci numbers generator have an IVF of below 1%. 50% of the static instructions have an IVF of 99% or 100%, and 43.8% have an IVF in the range 95%-99% (see Figure 2).

The highest performance improvements achieved by IVF-SID are obtained for Image Addition (58.7% to 77.4%) and Matrix Multiplication (52.2% to 77.7%). In Image Addition, 55% of the static instructions have an IVF of below 1%, and 45% have an IVF of above 99%. In Matrix Multiplication, 67.7% of the static instructions have an IVF of below 1%, and 22.6% have an IVF of above 99%. Both these benchmarks produce many independent elements in the output. When one of them is damaged, it often corrupts less than 1% of the whole output, which is the used IVF_{thr} value. For large images with millions of pixels which are common nowadays, one wrong pixel will most probably not even be visible.

Figure 4 shows that IVF-SID improves the performance of the ADPCM encoder much more than the other full applications. This is due to the fact that the encoding function in the ADPCM encoder, to which IVF-based ILCOFT is applied, is a substantial part of the whole application. In contrast, the IVF-enabled *forward_DCT* function in the JPEG encoder is only a part of the whole application, which also calls many other functions. Thus, the ADPCM encoder (and also decoder) functions affect the whole application performance much more than the functions in the JPEG encoder and decoder to which IVF-based ILCOFT is applied. The ADPCM encoder, however, has significantly more instructions with an IVF below IVF_{thr} (75.4% of the instructions, compared to 26% in the decoder), and thus it achieves a much higher performance improvement.

Figure 4 indicates that in most cases, the performance recovered by IVF-SID drops when the number of integer ALUs (and the fetch/decode/issue width) increases. This is due to the larger amount of computational resources available. The ILP introduced by instruction duplication is utilized more effectively on systems with more integer ALUs. Thus, the reduction of the number of executed instructions due to IVF-SID does not affect the performance so much. In contrast, for Image Addition and the JPEG Encoder, IVF-SID recovers less performance on a system with two integer ALUs than with four of them. We attribute this to the issue width, which is insufficient and becomes a bottleneck in these cases. Increasing the fetch/decode/issue width from two to four (while the number of integer ALUs is still two) increases the recovered performance from 58.7% to 70.3% for Image Addition.

Table 2: Fault coverage statistics.

Benchmark	Detected (%)	Wrong output (%)	Escapes (%)	Max. output damage (%)	Min. output damage (%)	Average output damage (%)
Image Addition	43.16	48.42	8.42	0.06	0	0.05
Matrix Multiplication	59.14	18.28	22.58	0.25	0.25	0.25
Fibonacci	95.92	0	4.08	0	0	0
SAD	95	0	5	0	0	0
JPEG Encoding	76.77	0	23.23	0	0	0
JPEG Decoding	10	78	12	0.03	0	0.01
ADPCM Encoding	33.33	31.31	35.35	0.64	0	0.12
ADPCM Decoding	87.76	0	12.24	0	0	0

Note that the IVF values distribution in static application instructions cannot be expected to always perfectly indicate how useful the IVF-based ILCOFT will be for the program. This is because performance depends on the dynamic behavior, on how many times every static instruction with a certain IVF will be executed.

3.4 Fault Coverage

To evaluate the fault coverage of IVF-SID with IVF_{thr} equal 1%, fault injection experiments have been conducted. Every benchmark has been run around 100 times, with one fault in the output of a random instruction within the kernel to which IVF-SID is applied. To simulate the worst-case scenario, burst (multi-bit) faults are injected rather than single-bit ones. This is achieved by changing instruction output to a random value. There are two reasons to inject only one fault per simulation. First, the current and near-future fault rates are not very high, and thus, more than one fault in a relatively small kernel is unlikely. Second, when injecting multiple faults per experiment, the chance that at least one of them will be detected increases. As a result, other faults that would not have been detected alone are hidden, and their effect on the output is not investigated.

Table 2 depicts the collected statistics. The second column shows the percentage of faults detected by instruction duplication. The third column indicates the percentage of simulations finished with wrong output (with an undetected fault). The fourth column shows the percentage of escapes, that is, undetected faults that did not propagate to the output. The subsequent columns demonstrate the maximum, minimum, and average observed output corruption percentage due to undetected faults.

Table 2 demonstrates that no undetected faults damaged more than 0.64% of the output. For most benchmarks, the maximum output damage is even smaller. The maximum output corruption of 0.06% in Image Addition means that in the output image, 1152 pixels out of almost 2 million are wrong. A visual inspection of the output image did not reveal any difference with the correct image. We believe that the observed output damage in other benchmarks is also tolerable.

SAD, Fibonacci, JPEG encoder and ADPCM decoder do not have any output damage due to undetected faults. All faults in these benchmarks were either detected or did not propagate to the output (escapes). These benchmarks have only a few instructions with low IVF values (see Section 3.3 and Figure 2). Most of their instructions have high IVF values, thus they are duplicated in IVF-SID, and faults are detected. This high instruction coverage results, however, in a smaller performance improvement of IVF-SID compared to full instruction duplication (see Figure 4).

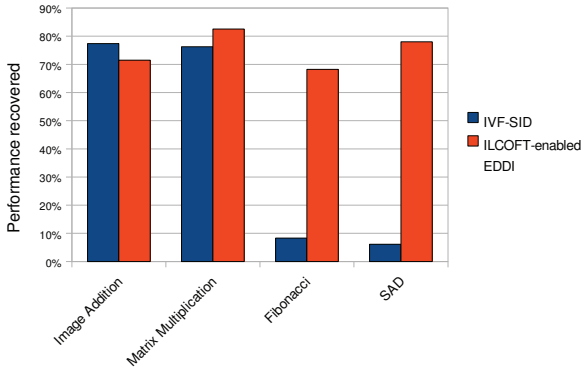

Figure 5: Comparison of performance penalty reduction achieved by IVF-SID over instruction duplication and by ILCOFT-enabled EDDI over EDDI.

3.5 IVF-SID Compared to ILCOFT-Enabled EDDI

In this section we compare IVF-SID to ILCOFT-enabled EDDI [4]. Error Detection by Duplicated Instructions (EDDI) [13] is a software error detection technique which duplicates all instructions in the program assembly code and inserts checks to determine if the original instruction and its duplicate produce the same result. ILCOFT-enabled EDDI duplicates only the critical instructions. In [4] instructions are categorized manually by a programmer, and are considered critical if they affect the control flow.

Figure 5 compares the amount of performance recovered by ILCOFT in IVF-SID and in ILCOFT-enabled EDDI. Table 3 compares the maximum and average output damage observed when these techniques are used. ILCOFT-enabled EDDI data is taken from [4].

Note that this is not a straightforward comparison. First, these techniques are very different, one is automatic and works in hardware, while the other is manual and is applied in software. Second, the processor organizations used in this work and in [4] differ in some parameters such as memory access latency, cache sizes etc. Both organizations, however, have an issue width of four and four integer ALUs.

The main message Figure 5 and Table 3 illustrate is that IVF-based ILCOFT is able to compete with a manual ILCOFT method. This is clear from the Image Addition and Matrix Multiplication performance and fault coverage: IVF-SID slightly improves performance of Image Addition and reduces its average output damage, and reduces performance of Matrix Multiplication and improves

Table 3: Output damage of IVF-SID and ILCOFT-enabled EDDI.

	IVF-SID		ILCOFT-enabled EDDI	
	Maximum	Average	Maximum	Average
Image Addition	0.06 %	0.05 %	0.13 %	0.01 %
Matrix Multiplication	0.25 %	0.25 %	99 %	3.1 %
Fibonacci	0	0	96.67 %	38.23 %
SAD	0	0	100 %	100 %

its fault coverage. For Fibonacci and SAD, however, IVF-SID has a serious performance disadvantage. This is due to the fact that ILCOFT-enabled EDDI considers only instructions affecting the control flow to be critical. This leads to many undetected faults causing an average output corruption of 38.2% for Fibonacci and 100% for SAD. IVF-SID does not allow any undetected faults in these two kernels to reach the output, because it protects all the vulnerable instructions. Thus, IVF-SID is more suitable if reliability is the primary concern, and ILCOFT-enabled EDDI is preferable if performance is more important. Furthermore, another advantage of IVF-SID is that it does not require the programmer to assign the necessary protection level to every assembly instruction manually, which demands a serious effort and is very error-prone.

4. CONCLUSIONS

This work introduces the concept of Instruction Vulnerability Factor (IVF). IVF determines how much of the final application output is corrupted due to faults in particular instructions, and can be estimated using fault injection experiments. It is shown that in most applications, instructions have different IVF values. Depending on the nature of the application, it may have a large number of instructions with low IVF values, which means that faults in these instructions are tolerable. Many applications in the multimedia domain, for example, have such characteristics. This work proposes to use the Instruction-Level Fault Tolerance Configurability (ILCOFT) principle based on the application profiling data containing every instruction's IVF value. Instructions with higher IVF values are protected better than instructions with lower IVF values.

A selective hardware instruction duplication scheme controlled by the instruction's IVF values (IVF-SID) is evaluated from the performance and fault coverage points of view. It is shown that, like in other ILCOFT schemes, both performance and fault coverage of IVF-SID depend on the nature of the application. Like other ILCOFT schemes, IVF-SID recovers maximum performance (and also saves energy) in applications that produce many independent output elements. For these applications, faults in many instructions are likely to affect only a small part of the final output, and thus many instructions have small IVF, and are not protected. The minimum performance is recovered in applications producing a single output value, or a set of dependent values. Any fault propagating to the final output corrupts it completely in these applications, thus most of the instructions need to be protected. On the other hand, these applications have a better fault coverage, because most instructions are protected.

The experimental results demonstrate that for both these types of applications, IVF is a useful measure which allows to determine the optimal way to balance the performance and fault coverage. Moreover, IVF can be estimated automatically. It releases application developers from the need to attribute code for ILCOFT manually. This saves time, effort, and improves reliability, because manual instruction FT level assignment is very error-prone. IVF-based ILCOFT achieves performance improvement comparable to a manual ILCOFT method. In addition, it provides much more accurate results than other automatic (compiler-based) instruction attribution methods, which are based on very general assumptions (that only instructions affecting the control flow should be protected).

For future work, we plan to enhance our experimental setup. Currently there is the following limitation. Both in IVF estimation and fault injection, only the first dynamic occurrence of every static instruction is processed, because it is unknown if this instruction will ever be executed later. In most cases this is not problematic. In rare cases, however, this can lead to inaccurate IVF estimation. For example, consider an instruction which calculates a condition for a branch controlling a loop. If it produces any non-zero value, the subsequent branch is taken (or not taken). At one loop iteration this instruction produces a 1, which becomes any non-zero number due to a fault, and the branch still performs correctly. However, at another loop iteration, this instruction produces a 0, which becomes a non-zero value due to a fault. In the latter case the branch behaves in the wrong way. Taking into account the dynamic application behavior would solve this problem when estimating IVF.

A possible future profiling procedure enhancement is to use a timer to detect cases when faults lead to a significant performance degradation. During IVF profiling, we have observed several cases when due to a fault, a critical loop iterates many orders of magnitude more times than it is supposed to. This leads to a corresponding performance degradation. If such performance degradation is unacceptable or if a significant profiling duration increase is unacceptable, a timer can be used to detect these cases. If the application execution takes more time than permitted, it can be terminated, assigning the maximum output damage to the resulting IVF value.

Both problems mentioned above can be solved by the following approach. All the instructions affecting the control flow can be assigned the maximum IVF, because they can always lead to application crashes. Then, the IVF is estimated using profiling only for the remaining instructions. The instructions affecting the control flow can be identified by the compiler.

IVF estimation as presented in this work is suitable for applications with equally important output values. However, there exist applications whose output consists of different parts of varying importance. For example, in a video sequence, bytes defining individual pixel values are less important than bytes defining frame attributes. For such applications the IVF estimation procedure has to be adapted to take into account the importance of individual output elements. This can be achieved by assigning different weight to the corruption of more and less important output elements.

5. REFERENCES

[1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. 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*, Washington, DC, USA, 2002, pp. 389–398.

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

[3] D. Borodin, B. Juurlink, and S. Vassiliadis, "Instruction-Level Fault Tolerance Configurability," in *IC-SAMOS VII: Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2007, pp. 110–117.

[4] D. Borodin, B. Juurlink, S. Hamdioui, and S. Vassiliadis, "Instruction-Level Fault Tolerance Configurability," *Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 89–105, October 2009.

[5] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient Fault Tolerance in Multi-Media Applications through Selective Instruction Replication," in *WREFT-08: Proc. of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*. New York, NY, USA: ACM, 2008, pp. 339–346.

[6] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *MICRO-36: Proc. of the 36th Annual IEEE/ACM Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 29.

[7] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[8] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *Proc. IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207–215, Nov 1995.

[9] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, ser. Annals of Mathematics Studies. Princeton, NJ: Princeton University Press, 1956, vol. 34, pp. 43–98.

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

[11] "Fibonacci numbers at Wikipedia," http://en.wikipedia.org/wiki/Fibonacci_number.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *MICRO-30: Proc. of the 30th Annual ACM/IEEE Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar 2002.