Run-time Coarse-Grained Hardware Mitigation for Multiple Faults on VLIW Processors

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys, Emmanuel Casseau

Run-time Coarse-Grained Hardware Mitigation for Multiple Faults on VLIW Processors

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys, Emmanuel Casseau
Univ Rennes, INRIA, CNRS, IRISA, France
Email: {emmanuel.casseau, angeliki.kritikakou}@irisa.fr

***Abstract*—As transistors scale down, processors are more vulnerable to radiation that can cause multiple transient faults in function units. Rather than excluding these units from execution, performance overhead of VLIW processors can be reduced when fault-free components of these affected units are still used. In the proposed approach, the function units are enhanced with coarse-grained fault detectors. A re-scheduling of the instructions is performed at run-time to use not only the healthy function units, but also the fault-free components of the faulty function units. The scheduling window of the proposed mechanism is two instruction bundles being able to explore mitigation solutions in the current and the next instruction execution. Experiments show that the proposed approach can mitigate a large number of faults with low performance and area overheads.**

I. Introduction

The energy transferred by radiation is known to cause transient faults on system hardware, which may severely affect the application execution. As technology size decreases, the hardware becomes more and more susceptible to radiation. Although multiple simultaneous faults have been neglected for a long time, they can no longer be negligible for technologies of 130nm and beyond [1]. A single particle hitting the silicon creates secondary particles, which can be emitted in several directions, and, thus, affect different nodes of a circuit [2].

Approaches to deal with multiple faults either apply hardware redundancy or software redundancy. Hardware redundancy inserts spare resources to the original processor to execute in parallel the same instruction and to compare the obtained results [3]. Small performance overhead is usually observed (due to the need of comparison), whereas the area overhead is significant. Software redundancy modifies the program by inserting redundant instructions to be executed on the original processor. Although the area is not increased, the impact on the execution time is significant due to the execution of redundant and comparison instructions [4]. However, when the system processor has parallel Function Units (FUs), the idle FUs can be used to execute these instructions. Such type of processors are the Very Long Instruction Word (VLIW) processors, which usually consist of complex FUs able to execute all types of operations and simpler ones that cannot execute sophisticated operations, such as multiplications and divisions. Existing approaches on VLIW processors that detect the faulty units before the application execution, such as schedule multi-versioning [5] and instruction modification [6], cannot be applied for soft errors. Few approaches detect the faulty FUs during execution. However, instruction duplication and re-execution approaches [7] and run-time rescheduling mechanisms [8] have not been explored for multiple errors. The work in [9] applies duplication and triplication of the instructions for multiple errors, whereas a mechanism is proposed to exclude the faulty FUs permanently for the rest of the execution. However, the negative impact on performance is significant.

To reduce this performance degradation while keeping the area overhead low, we propose a coarse-grained hardware mechanism that detects the faults during execution and excludes only the faulty components of the affected FUs. Our main contributions are: i) extension of the FUs with Built-In Current Sensors (BICS) [10], which monitor the induced transient currents to detect a fault; ii) the run-time instruction scheduling that excludes only the faulty components of the affected FUs exploring mitigation solutions in the current and the next instruction execution; and iii) extensive fault injection simulations varying the number of total and concurrently occurring faults. The obtained results show that the proposed approach mitigates several multiple faults with low overheads.

The organization of this paper is as follows. Section II presents an overview of the approach and the architecture of the proposed coarse-grained mechanism is detailed. Section III presents the experimental results and Section IV concludes this work.

II. Proposed Run-time mitigation mechanism

We use the 4-issue heterogeneous VLIW data-path of Fig. 1 to schematically illustrate our approach. In this example, the VLIW consists of a 3-stage pipeline with Fetch (F), Decode (DC) and Execute/Memory/Write-Back (EX/MEM/WB). A number of instructions is formatted as one big instruction, named as *instruction bundle*, which is executed in parallel by the FUs of the processor. The first issue is configured with 1 Arithmetic Logic Unit (ALU) and 1 Branch unit (BR), the second issue with 1 ALU and 1 Memory unit (MEM), and the third and fourth issues with 1 ALU and 1 Multiplication unit (MUL). The components in gray color correspond to the basic architecture and the components with green color are the extra hardware components required by our approach. The proposed approach focuses on multiple soft faults occurring in the arithmetic FUs, as they have the largest area footprint of the combinatorial components (see section III.A)). The faults in the storage components, e.g. register file, memory and pipeline registers, are assumed to be protected with other methods, such as Error Correction Codes (ECC).

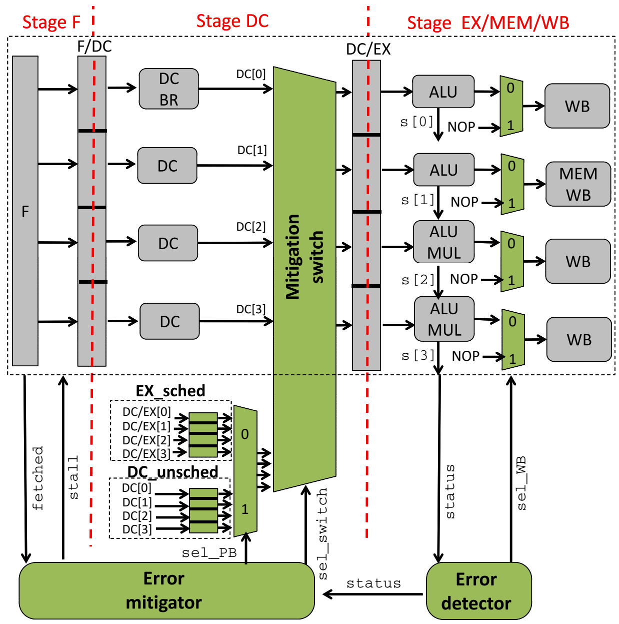

Fig. 1: Extension of the proposed mechanism on the original VLIW (4-issue VLIW example).

A. Overview and motivation

We will use the following example to present the idea of our mechanism. Fig. 2(a) depicts the original schedule. The instruction bundle is made of 3 additions (ADD_1, ADD_2, ADD_3) and 1 multiplication (MUL_1). To be able to detect an error, existing software approaches require to duplicate the instructions, such as in Fig. 2(b) and compare the results, and, thus, insert additional time slots increasing the application execution time. Existing hardware approaches are applied for permanent errors, whereas error detection is performed before the application execution, such as [6]. In case an error is detected, for instance if a fault occurs in the MUL unit of the third issue, hardware approaches insert an additional time slot and re-execute the instruction scheduled in the third issue to another compatible FU of another issue, as depicted in Fig. 2(c).

To reduce the negative impact on performance of existing approaches, we propose a mechanism that removes the need of replicated instructions and explores at run-time the re-scheduling of the faulty instructions to explore the healthy and idle FUs of the current bundle and the upcoming bundle, as depicted in Fig. 2(d). In this example, the instruction ADD_3 is moved to the third issue whose ALU is still healthy, whereas the instruction MUL_1 is moved to fourth issue. Therefore, no need exists for an additional time slot in this example.

B. Error detector

The error detector keeps the faulty status of the FUs and takes care of miscalculated results. Each FU is internally

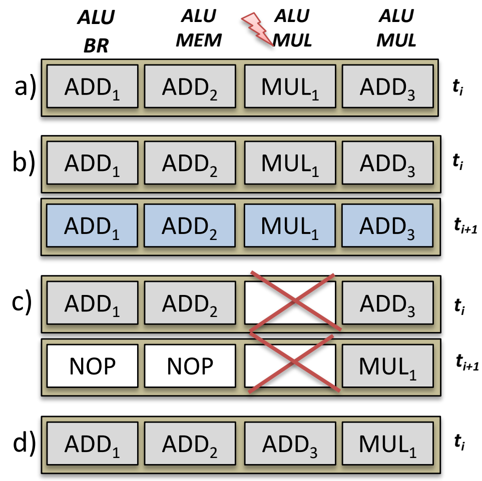

Fig. 2: Example: Motivation and overview of the approach.

enhanced with BICS sensors [10], [11]. A BICS is attached to a group of transistors. During normal operation, the current in the bulk of these transistors is approximately zero. Only the leakage current flows through the biased junction, which is still very low compared to the current generated by energetic particles. When an energetic particle generates a current in the bulk, the bulk-BICS captures that a transient fault occurs. The bulk-BICS has a reset mechanism that allows the fault detection to be active only as long as it takes to dissipate the transitory energy pulse. When the fault has vanished, the affected transistors can be used once again. A signal from the BICS from each FU of one issue is combined into a status signal, `s[i]` in Fig. 1, with a size equalling to the number of FUs of the issue i. Hence, if a bit in the signal `s[i]` is set, it means the corresponding FUs is affected by a fault. In case of one or more faults at cycle $k-1$, the results of the corresponding instruction(s) of the execution stage are miscalculated, and, thus, they must not be committed. For this purpose, each VLIW issue is enhanced with a multiplexer controlled by the signal `sel_WB` (with a size equalling to the number of VLIW issues) computed by the error detector. When a bit in `sel_WB` is set, the corresponding multiplexer passes a NOP result (instead of the miscalculated result) and the WB and MEM enable of the corresponding issue is disabled.

C. Mitigation switch

At cycle k, the *DC_unsched* shadow register stores the decoded instructions that couldn't be scheduled and the *EX_sched* shadow register keeps the instructions executed but in which an error occurred during their execution. Therefore, at cycle k the instructions to be scheduled can potentially come from three inputs: 1) the decoded instructions at cycle $k-1$ (*DC*) 2) the remaining instructions not scheduled at cycle $k-1$ (*DC_unsched* register) and 3) the executed instructions with a fault at cycle $k-1$ (*EX_sched* register). A mitigation switch is required to move at run-time the instructions in different issues than the ones defined by the compiler's original schedule at compile time. We reduce the complexity of the mitigation switch by performing hierarchical decisions. The first decision is whether a shadow register must be used or the currently decoded instructions and the second decision is which shadow register should be used (*EX_sched* or *DC_unsched*).

D. Error Mitigator

The error mitigator extracts from the fetch stage the required information and the dependencies between bundles and decides which of the three potential inputs that provide instructions for execution (*DC*, *DC_unsched*, and *EX_sched*) will be scheduled based on the status of the FUs and the type of the instructions.

The mitigator initially performs an early decoding of the instructions at the fetch stage to find the opcode, the destination registers and the source registers of each instruction and to identify instruction dependencies between two consecutive bundles.

To perform the mitigation process, each potential instruction input to the mitigation switch is modeled by a bit mask. A bit mask corresponds to the instruction scheduled at position i, as depicted in Fig. 3. a) *bit 3:* when this bit is set, the FU component required for the instruction execution is an ALU FU; b) *bit 2:* when this bit is set, the FU component required for the instruction execution is a MUL FU; c) *bit 1:* when this bit is set, it means the instruction is an instruction from a previous bundle that has not been scheduled yet; and d) *bit 0:* when this bit is set, it means the instruction has a dependency with at least one of the instructions of the next bundle. Then, each of the three potential inputs of the switch is represented by an array of bit masks (*EX_sched_mask*, *DC_unsched_mask* and *DC_mask*) that has a size equalling to the number of VLIW issues.

The status of all FUs components is represented by the array *status*, where each element is the `s` signal of an issue enhanced with an additional bit that is set when the issue is occupied as shown in Fig. 3 (*bit 0*).

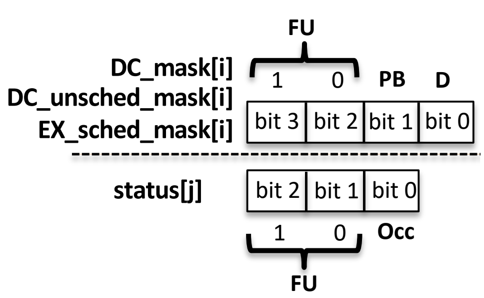

Fig. 3: Bit masks to model the inputs and FU component status of the mitigation switch.

The mitigator decides which of the three potential inputs is to be scheduled next based on the fault occurrences, as depicted by state machine diagram of Fig. 4, where $i \in [0, n]$ and n is the number of issues.

When at least one new fault occurred at cycle $k-1$ (`NF`), the executed instructions on the faulty FUs at cycle $k-1$ (which reside in *EX_sched* register) have to be executed again at cycle k.

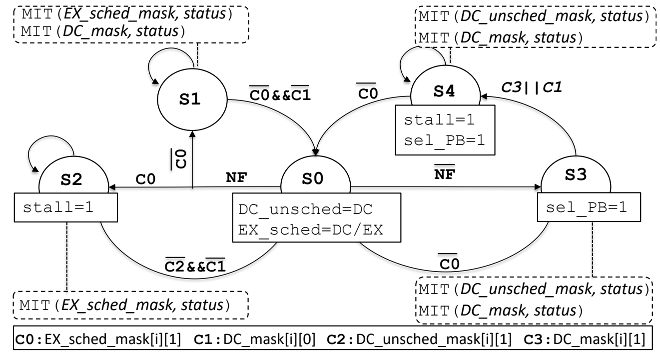

Fig. 4: Functionality of error mitigator.

(S1): If the faulty instructions are decoded instructions at cycle $k-2$ (`C0`==0), no stall is required. The array *EX_sched_mask* is used by the mitigation process, so as to schedule these instructions for execution at cycle k. Then, the decoded instructions at cycle $k-1$ are explored by passing the array *DC_mask* to the mitigation process.

(S2): If the faulty instructions are coming from a previous decoded bundle at cycle $k-3$, (`C0`==1), then the `stall` signal is activated and a new cycle is inserted for the re-execution of these instructions. The array *EX_sched_mask* is used as an input to the mitigation process. The process is repeated until all instructions are executed.

(S3): If no new fault occurred at cycle $k-1$, the mechanism schedules first the remaining decoded instructions from cycle $k-2$ for execution at cycle k. To do so, the array *DC_unsched_mask* is used as input to the mitigation process. Then, the current decoded instructions *DC_mask* are used as input to the mitigation process.

(S4): If there are still remaining instructions from the previous bundle and/or if there is any dependent instruction in current decoded instructions that cannot be scheduled, the `stall` signal is activated to stall the fetch and the decode stage for one cycle, so as to schedule these instructions.

The inputs of the mitigation process are: 1) the mask of the input signal to be scheduled, *input*; and 2) the array that represents the status of the FU components, *status*. The outputs are: 1) the control signal *sel_switch* of the mitigation switch; 2) the control signal *sel_PB* of the multiplexer between the shadow registers; 3) the updated mask of the input signal; and 4) the updated *status*. The procedure is given by Alg. 1. For all the instructions i of the *input* (line 3) and for each issue j described by *status* (line 4), if the instruction i has not been scheduled and the j issue is unoccupied (line 5), check if the required FU is available (line 6). If this is true, the occupied bit of the corresponding *status* is set (line 7), the remaining bit of the *input* is cleared, since the instruction is scheduled (line 8), and the signal *sel_switch* instructs the switch to pass the instruction currently at issue i to issue j (line 9) and the next instruction is explored.

Algorithm 1 Mitigation process

Inputs: $input, status$
Outputs: $input, status, sel\_switch$
for $i \in \{0, n\}$ **do** ▷ for each instruction
  for $j \in \{0, n\}$ **do** ▷ for each issue
▷ if the instruction is not scheduled and the issue is free
    if $(input[i][1] \&\& \overline{status[j][0]})$ **then**
▷ if the required FU is healthy
      if $(input[i][3] \&\& \overline{status[j][2]} \mid\mid input[i][2] \&\& status[j][1])$ **then**
▷ The issue is occupied
        $status[j][0] = 1;$
▷ The instruction is scheduled
        $input[i][1] = 0;$
▷ Instruction at position i pass to issue j
        $sel\_switch[i] = j;$
        $break;$
      end if
    end if
  end for
end for

III. Evaluation results

The original processor and the processor enhanced with the proposed approach have been developed in C++ and synthesized using the Catapult High Level Synthesis (HLS) tool to obtain the RTL design. The gate-level netlist was generated by the Design Compiler tool from Synopsys using 28 nm ASIC library. To evaluate our approach, we use ten benchmarks from the MediaBench suite [12]. The benchmarks are compiled with VEX compiler [13] for each configuration. For the evaluation results, we used the VEX VLIW processor architecture [14] with two different configurations:

- 4-issue VLIW configured with 4 ALU FUs, 2 MUL FUs, 1 MEM FU and 1 BR FU,
- 8-issue VLIW configured with 8 ALU FUs, 4 MUL FUs, 2 MEM FUs and 1 BR FU.

For every benchmark, Table I shows the average number of useful instructions per bundle, ILP, for both the 4-issue and 8-issue VLIW configurations. Because ILP inside bundles is low compared to the number of issues, idle issues of the current or next bundles can be used to execute the faulty instructions in addition to switching instructions with healthy components of units affected by a fault.

A. Area analysis of the unprotected VLIW processor

First of all, we present the logic area of each unit of the original unprotected VLIW architecture (as depicted in gray color in Fig. 1) in Table II and the area of each pipeline stage for the 4-issue and 8-issue VLIW in Table III. The results motivate the focus of the proposed approach on the execute stage of the VLIW processor, since it covers more area, and, thus, it is more likely to be exposed to faults.

TABLE I: Benchmark characteristics

Benchmark	ILP 4-issues	ILP 8-issues
adpcm_dec	1.77	2.28
adpcm_enc	1.82	2.41
bcnt	2.49	3.62
fft32x32s	2.85	4.19
huff	1.51	1.75
motion	1.94	2.39
dct	2.22	3.31
crc	1.76	1.8
fir	2.09	2.5
matrix_mul	2.61	4.46

TABLE II: Area of main VLIW combinatorial units (μm^2).

DC	BR	ALU FU	MUL FU	MEM/WB
250	2,290	1,533	2,310	358

B. Performance

In this paper, we present first experiments to evaluate the benefit of our approach based on coarse-grained FU exploration. The fault model is simplified as much as possible : we randomly injected multiple faults during the benchmarks' execution and, in order to reduce the number of scenarios, we consider the faults as persistent after they occur, i.e. they last for the rest of the execution. Performance overhead is thus the upper bound compared to a transient fault based simulation. Each benchmark is tested for: i) 0 up to 4 multiple faults for the 4-issue configuration; and ii) 0 up to 10 multiple faults for the 8-issue configuration [1].

Table IV and Table V show the number of execution cycles required to execute the benchmarks considering :

- the original application on the non-protected VLIW (Original),
- the proposed run-time coarse-grained mitigation approach, i.e a time and space idle slot exploitation method in addition to current sensors to use healthy components of faulty function units
- the version where only detection can occur through time and space idle slot exploitation with duplication of the instructions during the execution (2I),
- the version where correction and masking occur through time and space idle slot exploitation with triplication of the instructions during the execution (3I),
- the approach where correction and masking occur by triplicating the FUs in each issue with respect to the original unprotected processor while running the original application (3FUs).

The performance results are obtained by taking the mean value of 20 simulations running the same benchmark and the faults are injected at random execution cycles for each simulation. When no faults occur, the proposed approach have the same performance as the unprotected version, i.e. the number of execution cycles is the same as the original execution. When

[1] 4 and 10 faults for the 4-issue and 8-issue VLIW configurations respectively, is the maximum number of faults that the approach can sustain at the same time

TABLE III: Area of VLIW stages (μm^2).

Stage	4-issues	8-issues
DC	3,290	6,290
EX	10,752	21,5094
MEM/WB	1,432	2,864

detection and correction occurs by triplicating the FUs in the issues, the number of execution cycles is the same as the original version one (not presented in Table IV and Table V).

From the Table IV, we observe that the average performance overhead of the proposed approach for the 4-issue configuration is 0% for zero fault up to 83% for four faults. For the 2I (3I) approach, the average performance overhead is 72% (152%) for zero fault up to 262% (439%) for four faults. From the Table V, we observe that the average overhead of the proposed approach for the 8-issue configuration is 0% for zero fault up to 141% for ten faults. For the 2I (3I) approach the average overhead is 38% (104%) for zero fault up to 378% (626%) for ten faults.

Overall, we observe that: 1) the proposed approach inserts significantly lower performance overhead than the approaches with instruction duplication or triplication and 2) in several benchmarks, when the number of faults is low, our approach's performance is very close to the original one, i.e. without faults. The obtained gain of the proposed approach comes from the fact that whenever a persistent fault is detected, the proposed hardware mechanism exploits healthy FUs in the current and the next bundle execution in addition to idle FUs.

C. Area and delay overhead

Table VI presents the impact in area and in delay of each component of the proposed run-time coarse-grained mechanism for the 4-issue VLIW. It should be stressed that the delay introduced by the error mitigator does not affect the clock of the VLIW processor, since it is executed in parallel with the VLIW data-path.

TABLE VI: Area and delay overheads introduced by the the components of the proposed mechanism for 4-issue VLIW.

Component	area (μm^2)	delay (ns)
Mitigation Switch	3,918	0.16
Error mitigator	2,112	2.44
Error detector	3,268	0.84

Table VII shows the area overhead of the proposed mechanism, the approach that provides error masking by applying triplication of instructions and the approach that triplicates the FUs at each issue. Compared to the unprotected version, the proposed approach implies an area overhead of 18% and 28% for the 4-issue and for the 8-issue configurations respectively. For the 3I approach the area overhead is 24% for the 4-issue configuration while for the 8-issue configurations it is 30%. It is quite similar to the area overhead of the proposed approach. However the proposed approach achieves much better performance as seen in previous section. For the 3FUs approach the area overhead is 45% for the 4-issue while for the 8-issue configurations it is 56%.

TABLE VII: VLIW processor area footprint.

Approach	4-issues		8-issues	
	area (μm^2)	area (%)	area (μm^2)	area (%)
Original	50,844	-	79,661	-
Instr. triplication (3I)	62,812	23.54	103,598	30.05
FUs triplication (3FUs)	73,523	44.60	124,137	55.83
Proposed mechanism	60,143	18.29	102,201	28.3

IV. Conclusion

A hardware mechanism is proposed for multiple soft faults that characterizes the FUs of the VLIW processor and reschedules at run-time the instructions, exploiting in time and space idle FUs and healthy components of faulty function units so as to mitigate faulty instructions. From the obtained results, it is shown that several multiple faults can be mitigated with significant reduction in the performance and area overheads compared with approaches that duplicate and triplicate the instructions during execution or triplicate the FUs in the issues. Work in progress is about experiments with faults that last various durations.

References

[1] T. Heijmen, "Radiation-induced soft errors in digital circuits – a literature survey," 2002.

[2] V. Ferlet-Cavrois, P. Paillet and M. Gaillardin et al., "Statistical Analysis of the Charge Collected in SOI and Bulk Devices Under Heavy lon and Proton Irradiation - Implications for Digital SETs," *IEEE Transactions on Nuclear Science*, pp. 3242–3252, Dec 2006.

[3] J. Klecka, W. Bruckert, and R. Jardine, "Error self-checking and recovery using lock-step processor pair architecture," 2002.

[4] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance", *CGO*, pp. 243–254, 2005.

[5] R. Karri et al., "Computer aided design of fault- tolerant application specific programmable processors," *TC*, vol. 49, no. 11, pp. 1272–1284, Nov 2000.

[6] M. Scholzel and S. Muller, "Combining hardware- and software-based self-repair methods for statically scheduled data paths," *DFT*, pp. 90–98, Oct 2010.

[7] A. L. Sartor et al., "Exploiting idle hardware to provide low overhead fault tolerance for vliw processors," *JETC*,vol. 13, no. 2, pp. 13:1–13:21, Jan 2017.

[8] R. Psiakis, A. Kritikakou, and O. Sentieys, "Neda: Nop exploitation with dependency awareness for re- liable vliw processors," *ISVLSI*, pp. 391–396, July 2017.

[9] R. Psiakis, A. Kritikakou, and O. Sentieys, "Run-time instruction replication for permanent and soft error mitigation in vliw processors," *NEWCAS*, pp. 321–324, June 2017.

[10] R. Viera, R. Possamai Bastos, J. Dutertre, O. Potin, M.-L. Flottes, G. Di Natale, and B. Rouzeyre, "Validation of Single BBICS Architecture in Detecting Multiple Faults," *ATS*, Nov 2015.

[11] R. P. Bastos, J. M. Dutertre, and F. S. Torres, "Comparison of bulk built-in current sensors in terms of transient-fault detection sensitivity," *VARI*, pp. 1–6, Sept 2014.

[12] C. Lee et al., "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," *MICRO*, pp. 330–335, Dec 1997.

[13] Hewlett-Packard Laboratories. VEX Toolchain, http://www.hpl.hp.com/downloads/vex/.

[14] J. A. Fisher, P. Faraboschi, and C. Young, "Embedded computing: a VLIW approach to architecture, compilers and tools," Elsevier, 2005.

TABLE IV: Performance comparison for 4-issue VLIW

Benchmarks	**Original**	**Proposed mechanism**					**Instr. duplication (2I)**					**Instr. triplication (3I)**				
		0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
adpcm_dec	386	386	412	437	538	565	676	731	809	837	1124	996	1150	1153	1605	1682
adpcm_enc	409	409	462	469	503	611	722	831	862	902	1210	1062	1094	1230	1387	1817
bcnt	478	478	511	512	784	1149	870	904	904	1199	2308	1264	1297	1297	1989	3364
fft32x32	569	569	758	775	932	1360	1097	1290	1568	1687	2710	1654	1943	2052	2600	4002
huff	1101	1101	1136	1137	1177	1491	1668	1815	1895	1896	2830	2348	2349	2920	2920	4167
motion	344	344	350	392	619	621	548	549	656	716	1211	813	868	868	1110	1817
dct	1288	1288	1353	1626	1844	2607	2533	2662	3314	3428	5173	3779	3972	4962	5231	7752
crc	12228	12228	12274	12275	14851	20972	15978	15979	18896	31420	41431	22600	22646	22647	45847	61887
fir	6852	6852	8053	8594	8715	11595	11780	12802	14903	15564	23063	16890	20491	21031	24154	34520
mat_mul	11142	11142	15015	16039	16359	21597	20968	24841	25961	31084	43178	30795	34668	43884	47400	64733
average overhead		0.0%	12.2%	18.4%	41.6%	82.8%	72.5%	88.0%	111%	139%	262%	152%	175%	202%	277%	439%

TABLE V: Performance comparison for 8-issue VLIW.

Benchmarks	**Original**	**Proposed mechanism**										
		0	1	2	3	4	5	6	7	8	9	10
adpcm_dec	302	302	309	312	312	370	407	433	443	437	471	558
adpcm_enc	323	323	327	339	342	433	405	478	481	489	572	590
bcnt	333	333	334	335	335	336	544	525	674	545	664	1130
fft32x32	400	400	407	424	479	659	655	738	737	875	1385	1388
huff	951	951	946	946	1039	959	960	1039	1040	1078	1113	1466
motion	280	280	277	278	284	357	361	358	354	378	415	610
dct	872	872	881	898	968	1078	1335	1369	1388	1543	2573	2576
crc	11955	11955	11956	11956	11956	11957	12217	15105	12232	12516	15106	20972
fir	5709	5709	5890	5889	5892	6072	6854	6853	7750	8111	8708	11814
mat_mul	6533	6533	6534	6534	10619	6537	10630	10950	11717	14323	14833	20356
average overhead		0%	1.0%	2.2%	4.6%	17.1%	36.7%	42.5%	46.2%	62.9%	94.8%	141%

Benchmarks	**Original**	**Instr. duplication (2I)**										
		0	1	2	3	4	5	6	7	8	9	10
adpcm_dec	302	407	432	443	563	702	710	765	762	773	845	1116
adpcm_enc	323	439	470	478	492	496	756	769	848	897	866	1278
bcnt	333	547	548	549	549	936	549	933	938	1022	1226	2064
fft32x32	400	656	705	751	783	839	1015	1230	1391	1509	1778	2789
huff	951	1038	1039	1040	1075	1076	1610	1605	1764	1744	1830	2800
motion	280	361	383	383	551	384	558	558	561	655	715	1226
dct	872	1339	1432	1448	2450	2490	1776	2547	2685	2949	3429	5131
crc	11955	12230	12231	12231	12515	12517	16219	16216	16231	18908	41344	41382
fir	5709	6853	7754	7934	7935	8115	10515	13697	13812	14533	15488	23505
mat_mul	6533	10950	11719	11719	11719	19904	19899	20520	22076	22851	28517	40736
average overhead		38%	46%	48%	71%	96%	108%	143%	155%	174%	225%	378%

Benchmarks	**Original**	**Instr. triplication (3I)**										
		0	1	2	3	4	5	6	7	8	9	10
adpcm_dec	302	612	619	616	836	761	1037	1127	1154	1587	1295	1692
adpcm_enc	323	662	675	725	682	1049	1064	1204	1294	1280	1486	1899
bcnt	333	764	782	782	783	1389	781	1417	1413	1422	1954	3449
fft32x32	400	996	1029	1066	1085	1399	1830	1862	2077	4073	2745	4311
huff	951	1519	1520	1521	1533	1593	1635	2268	2274	2832	2828	4085
motion	280	496	497	502	597	603	807	790	828	1112	1117	1750
dct	872	2136	2225	2218	2506	3566	2727	3900	4024	4286	5149	7626
crc	11955	15962	15963	15978	15980	15980	16979	20019	22906	22914	60038	61808
fir	5709	10877	11179	11179	11360	16219	17121	19814	19939	20584	24064	35147
mat_mul	6533	16408	16409	16410	16411	29433	30912	30870	33218	33502	43379	61015
average overhead		104%	108%	110%	124%	200%	205%	259%	277%	362%	404%	626%