NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys

NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys
University of Rennes 1 - IRISA/INRIA, Rennes, France
rafail.psiakis@inria.fr

***Abstract*—Critical applications require reliable processors that combine performance with low cost and energy consumption. Very Long Instruction Word (VLIW) processors have inherent resource redundancy not constantly used due to application's fluctuating Instruction Level Parallelism (ILP). Reliability through idle slots utilization is explored either at compile-time, increasing code size and storage requirements, or at run-time only inside the current instruction bundle, adding unnecessary time slots and degrading performance. To address this issue, we propose a technique to explore the idle slots inside and across original and replicated instruction bundles reclaiming more efficiently the idle slots and creating a compact schedule. To achieve this, a dependency analysis is applied at run-time. The execution of both original and replicated instructions is allowed at any adequate function unit, providing higher flexibility on instruction scheduling. The proposed technique achieves up to 26% reduction in performance degradation over existing approaches.**

I. Introduction

To accomplish the todays increasing demands for performance, while preserving or even reducing area and energy consumption, processors follow technology and voltage scaling trends. However, decreasing the transistors size and voltage leads to increased vulnerability and thus high demands for reliability [1]. Some errors that may affect the processor operation are: soft errors caused by radiation, circuit aging-wearout induced errors, thermal induced errors, errors induced by electromagnetic interference, etc. [2]. To protect the processors against them, several reliability techniques have been developed using either Hardware (HW) or Software (SW) redundancy. HW redundancy approaches insert additional hardware resources to execute the instructions more than once [3]. They provide the best performance, almost equal to the performance of the unprotected version, but in cost of area overhead. SW redundancy approaches execute the replicated instructions on the same hardware resources increasing the execution time [4].

VLIW processors offer a large number of hardware resources that are not always used due to processor issue-width and the intrinsic ILP available in each application. The idle issue slots can be used to execute replicated instructions. Such approaches can be implemented in software or in hardware. Software implementation approaches duplicate/triplicate the code instructions, while inserting additional instructions to perform error detection/correction. Idle slots exploration and scheduling is performed by the compiler and the obtained result is usually a dense schedule [5]. However, the code size and the storage requirements are highly increased. To meet the limited system resources, hardware approaches perform the instruction replication and scheduling at run-time. However, existing approaches [6], [7] explore the idle slots only inside the current instruction bundle, without exploring next instruction bundles. As a result, unnecessary time slots are added even if the idle slots of the next bundle could be used, leading to a more spare schedule and to performance degradation.

To improve performance, while preserving reliability, the proposed approach performs idle slots (i.e. Nop instructions) Exploitation with Dependency Awareness (NEDA). Compared with existing approaches, NEDA explores the VLIW idle slots both inside and across instruction bundles leading to a more efficient utilization of the resources and a more dense schedule, similar enough to the one created by the software techniques. NEDA performs a run-time dependency analysis between consecutive instruction bundles. It takes advantage of the idle slots of the next instruction bundle and its potential replication instead of directly inserting additional time slots. In addition, NEDA does not limit the scope of VLIW slots, i.e. the FUs are not coupled. Therefore, the FUs can execute any original or redundant instruction of their type, providing higher flexibility on instruction scheduling and thus better utilization of the idle slots. The experimental results show a reduction in performance degradation up to 26%.

The remaining of this paper is organized as follows. Section II describes the proposed technique, Section III analyzes the approach modifications in architecture, Section IV presents the experimental results, whereas Section V discusses the related work. Finally Section VI concludes this work.

II. NEDA Approach

The target domain of the proposed approach is VLIW processors, where a number of instructions is formatted as one big instruction, named *instruction bundle*, which is issued in parallel to the pipelined Function Units (FU) of the processor.

VLIW	Very Long Instruction Word	RA	Resource Analyzer
ILP	Instruction Level Parallelism	CL	Control Logic
FU	Functional Unit	EI	Execution Intermediate
F	Fetch	CI	Commit Intermediate
DC	Decode	RAW	Read After Write
EX	Execute	WAR	Write After Read
M/WB	Memory/Write-Back	WAW	Write After Write
DA	Dependency Analyzer	FR	Future Register
CR	Commit Register	ERV	Error Reporter/Voter

The proposed methodology takes advantage of the idle issue slots inside and across VLIW instruction bundles to execute original and replicated instructions in order to provide fault tolerance. NEDA is not limited by: 1) the applied fault tolerant technique, since it is applicable for both duplication and triplication of the instructions supporting error detection and mitigation, and 2) the VLIW structure, since it is applicable for any issue width and number and type of FUs.

We will use the example of Fig. 1a to illustrate NEDA approach. Fig. 1a depicts an 8-issue width VLIW and two instruction bundles, B_i and B_{i+1}, from the original code. B_i has five instructions and three idle slots, while B_{i+1} has seven instructions and one idle slot. The light gray boxes depict the original instructions. Without loss of generality and to keep the illustration example simple, we use duplication of the instructions as the applied fault tolerant technique.

Fig. 1b shows the approach of existing techniques, similar to [6], where the dark gray boxes depict the duplicated instructions. The latter are executed only by the coupled pipeline. As the idle slots are explored only inside the current instruction bundle and the idle slots in B_i are not sufficient to execute all the replicated instructions, an additional time slot t_{i+1} has to be inserted (here for *instr5* and its replicate). The same holds for the second instruction bundle, B_{i+1} with an extra time slot t_{i+3}. Hence, each time the ILP of the instruction bundle is greater than half of the issue width or the required resources are not available, extra time slots have to be added increasing the execution time.

Fig. 1c depicts the proposed approach. For the current bundle B_i, NEDA analyses the instructions inside this bundle to calculate the number of existing idle issue slots and the available FUs. Four possible cases may exist for the scheduling of the replicated instructions of B_i taking into account the type and the number of the FUs:

a) Only current bundle is used: In this case, enough idle slots exist in the current bundle and the required number and type of FUs are also available. Hence, all replicated instructions are scheduled in the current time slot t_i.

b) Next bundle is also used: This case is valid when i) not enough idle slots exist, or ii) enough idle issue slots exist to schedule the redundant instructions, but the required type of FUs is not available. For instance, in Fig. 1a assuming that *instr1* and *instr2* of B_i are memory instructions and the VLIW is able to execute up to two memory instructions per bundle, some of the memory instructions (original or replicated) have to be scheduled on the next time slot, even if enough idle slots exist.

Compared with existing approaches, NEDA explores the next instruction bundle B_{i+1} to schedule instructions from the current bundle avoiding the insertion of additional time slots. More precisely NEDA explores B_{i+1} to find the existing idle slots and the available FUs. It calculates if enough resources exist to schedule the remaining instructions that do not fit in B_i. NEDA performs a dependency analysis between the instructions of the current bundle B_i and the ones of the next bundle B_{i+1}. The analysis is performed to give priorities

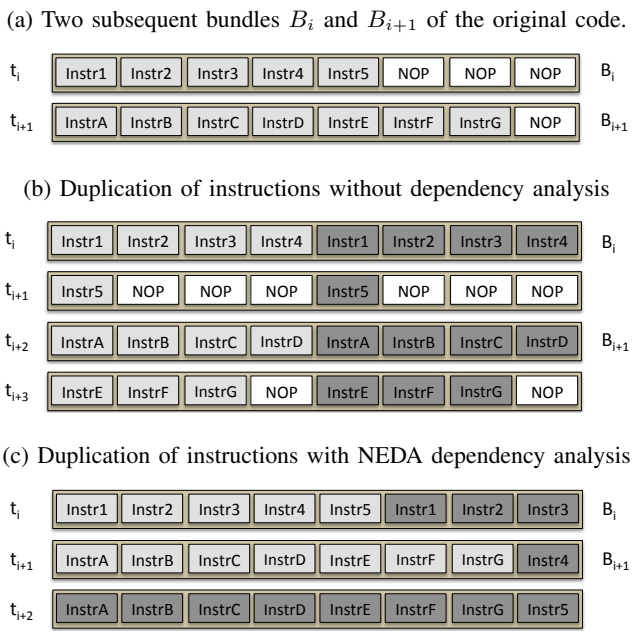


Fig. 1: Proposed approach illustrated on an 8-issue VLIW

to the scheduling of the instructions of B_i. The dependent instructions have increased priority, and, thus, they are scheduled in the idle slots of the current bundle B_i, whereas the independent ones can be scheduled in the idle slots of the next bundle. In Fig. 1a, assuming that *instr1*, *instr2* and *instr3* have dependencies with at least one instruction of B_{i+1}, they are prioritized against *instr4* and *instr5* and are scheduled in the current B_i.

c) Replication of the next bundle is also used: In case of large ILP or insufficient resources in B_{i+1}, NEDA predicts an additional time slot for B_{i+1}'s replicated instructions, thus the total idle slots are duplicated. Hence, NEDA explores the idle issue slots in this extra time slot to schedule the remaining instructions of the current bundle B_i. In Fig. 1c the first three replicated instructions *instr1*, *instr2* and *instr3* of B_i are scheduled in the available slots of the current bundle B_i and *instr4* and *instr5* remain to be scheduled in subsequent time slots. As B_{i+1} is almost filled with instructions, NEDA foresees an extra time slot t_{i+2} for B_{i+1} and uses the two idle slots in t_{i+1} and t_{i+2} to schedule *instr4* and *instr5* of B_i.

d) Otherwise: This case is valid when: a) The total number of idle slots or FUs in the current bundle, in the next bundle and in its replication is not enough to schedule all replicated instructions, or b) too many dependencies exist that prohibit the parallel execution of B_i and B_{i+1} instructions. Eventually no further exploration can be applied and NEDA inserts an extra time slot for the remaining unscheduled replicated instructions of the current bundle B_i.

III. Impact on Processor Architecture

To describe our methodology, a 4-issue VLIW architecture is shown in Fig. 2, which consists of a 4-stage pipeline with

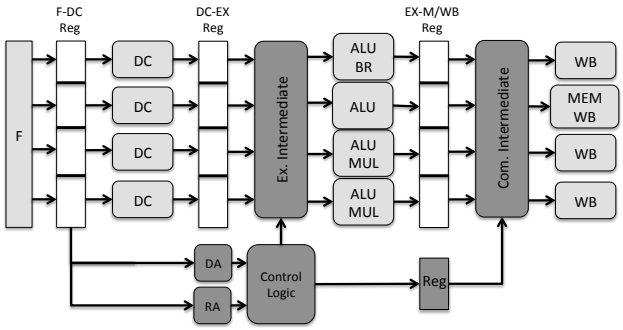


Fig. 2: NEDA's modified pipeline

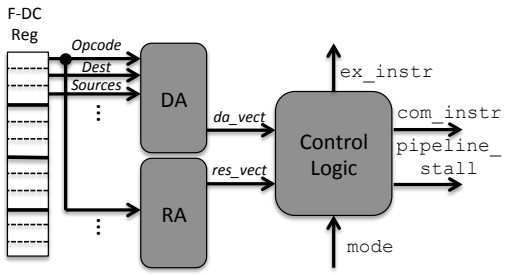


Fig. 3: Dependency/Resource Analyzer and Control Logic

Fetch (F), Decode (DC), Execute (EX) and Memory/Write-Back (M/WB). With gray color we highlight the additional components added by our approach. The Dependency Analyzer (DA) is the component responsible for analyzing two subsequent bundles in order to identify potential dependencies. The Resource Analyzer (RA) informs the Control Logic (CL) about the type and the number of the instructions. The Execution Intermediate (EI) passes the instructions currently decoded, their replicates and the ones left from previous bundles to the FUs. The Commit Intermediate (CI) performs error detection/correction and passes the instructions to the M/WB stage. The CL unit decides the instructions to be scheduled at the current slot using the analysis provided by DA and RA and configures accordingly the EI and CI. The next subsections describe each component in more detail.

A. Resource and Dependency Analyzers

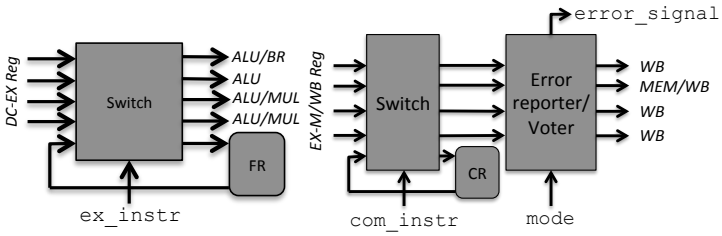
The RA, as depicted in Fig. 3, extracts each instruction's opcode and computes one vector per type of resources, res_vect_m, representing the instructions that are using the resources of type m. These vectors are concatenated to the output vector of RA, res_vect that informs CL about the type of required resources. The RA uses a buffer to store this vector for future analyzing. The DA performs dependency analysis between two subsequent bundles. Initially, DA takes the information from the bundle in the fetch stage by extracting each instruction's opcode, destination and source registers and uses a buffer to store this information. Because the architecture is pipelined, when the next fetched bundle is decoded, the DA can perform the dependency analysis between the two subsequent bundles and compute a dependency vector, da_vect. There are three possible dependency cases: 1) Read After Write (RAW), 2) Write After Read (WAR) and 3) Write After Write (WAW). RAW and WAW are taken care by DA, while WAR never occurs since it is prevented by architecture's design. WAR occurs if two instructions of the same bundle, one writing and the other reading the same register, are executed at different time slots. In NEDA this could not be the case. Although NEDA postpones the execution of the instructions, their decoding and, thus, all instruction register assignments happen at the correct time. DA's functionality for a bundle B_i can be expressed as:

$$dest(B_i[j]) = source(B_{i+1}[k]) \ \vee \ dest(B_{i+1}[k]) \\ \rightarrow da\_vect[j] = 1, \ \forall j, k \tag{1}$$

where B_i is the bundle with order number i and $B_i[j]$ is a specific instruction of this bundle with a position order number j. $B_{i+1}[k]$ is a specific instruction of the next bundle B_{i+1}. The $dest(B_i[j])$ and the $source(B_{i+1}[k])$ are the destination and the source registers of the instructions $B_i[j]$ and $B_{i+1}[k]$, respectively. Eq. 1 describes that an instruction $B_i[j]$ is dependent, if there is an instruction of B_{i+1} with a destination or source register that is the same as the destination register of $B_i[j]$. The $da\_vect[j]$ is the jth element in the output vector of DA which indicates whether the instruction $B_i[j]$ is dependent or not. For instance, if only $da\_vect[1] = 1$, the dependency vector is $da\_vect = \{0, 1, 0, 0\}$, which means that the second instruction of bundle B_i has one or more dependency conflicts with instructions of the next bundle B_{i+1}.

B. Execution and Commit Intermediates

The EI is inserted in the EX stage of the pipeline and it consists of a switch unit and a Future Register (FR), as depicted in Fig. 4a. The switch unit, fed by the DC-EX register and the FR, drives its inputs to the FUs of the execution stage according to CL configuration signal, named `ex_instr`. The FR keeps the instructions that are postponed for the next execution. The CI connects the execution results with the M/WB stage. As illustrated in Fig. 4b, the CI consists of a switch unit, a Commit Register (CR) and an Error Reporter/Voter (ERV). The switch unit, fed by the EX-M/WB register and the CR, drives its inputs to the ERV according to CL's configuration signal, named `com_instr`. The error reporter/voter implements the



(a) Execution Intermediate (b) Commit Intermediate

Fig. 4: Intermediate Components

detection/correction of the fault tolerance technique. The fault tolerance technique is a design choice between i) duplication of the instructions (DMR) and ii) triplication of the instructions (TMR), selected by the signal `mode`. Hence, it either compares instructions by two, commits one of the two results and reports an error through the signal `error_signal`, or compares instructions by three, votes and commits the correct result and reports an error. The CR supplies the switch with the instruction results kept from previous executions. This occurs if an instruction and its replicate are executed in different time slots, and, thus, they are not both available at the same time.

C. Control Logic unit

The state diagram and the flowchart in Fig. 5 represents CL's functionality. We use DMR as an illustration example, but similar logic is applied for the TMR. The CL operates following a 2 state FSM. The CL is in the first state, S0, every time a new bundle enters the fetch stage, while it passes to the second state, S1, when a new time slot needs to be added. On the first state, the CL calculates through condition C1 if instructions have to be postponed. The number of potentially postponed instructions is given by $\#post\_instr_m$ and computed by Eq. 2 for each type of resources, while w indicates the issue width.

$$\#post\_instr_m = 2 * \sum_{j=0}^{w} res\_vector_m(B_i[j]) + \sum_{j=0}^{w} res\_vector_m(FR[j]) - \#avail\_res_m \quad (2)$$

where $2 * \sum res\_vector_m(B_i)$ are the requirements in resources for the decoded instructions and their replicates, $\sum res\_vector_m(FR)$ the requirements for the instructions stored to FR, and $\#avail\_res_m$ the available m type resources of the architecture.

a) If the C1 condition is false: there is no need for an extra time slot, because all the instructions can be scheduled and fit to the current time slot. Then, the CL configures the EI's switch on how to pass these instructions to the available FUs. The command `ex_schedule` in Fig. 5 represents this functionality. The CI's switch is also configured on how to pass to the error reporter/voter: i) the instructions and their duplicates executed alongside, and, ii) the executed instructions that their duplicates' results are already in CR, as they have been executed in the previous execution. The error reporter/voter reports any detected errors and passes the results to the M/WB phase for commit. The command *com_schedule* represents this functionality.

b) If the C1 condition is true: the CL determines which instructions can be postponed according to the available resources and the dependency analysis. This information is stored to a vector, $post\_vector_m$. The $post\_vector_m$ of a bundle B_i has twice the size of the issue width to take into account the duplicated instructions of this bundle. When no dependency exists between the instruction $B_i[j]$ in B_i and the instructions of next bundle B_{i+1}, the element j (for the original instruction j) and the position $j+w$ (for the duplicate

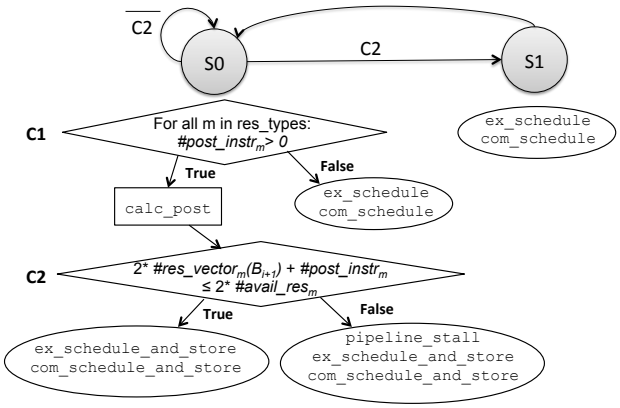


Fig. 5: Control Logic FSM and flowchart

of instruction j) of $post\_vector_m$ are set to 1 (Eq. 3). Then, the different type of resources are explored by giving priority to the limited availability of each type of resources m and to the already postponed instructions inside FR. The $constr\_vector_m$ (computed in Eq. 4) consists of a number of instructions equal to $\#post\_instr_m$ which can be currently scheduled.

$$\forall j : post\_vector_m[j], post\_vector_m[j+w] = \begin{cases} 0 & \text{if } da\_vector_j = 1 \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

$$cnstr\_vector_m = \{x : res\_vector_m(B_i[x]) = 1 \wedge |cnstr\_vector_m| = \#avail\_res_m\} \quad (4)$$

The $post\_vector$ is updated by removing the $constr\_vector$. This exploration is applied for all types of resources m. After that, the CL computes the number of the fetched instructions, $\#res\_vector_m(B_{i+1})$. It checks through the condition C2, if the number of the fetched instructions, their duplicates and the instructions need to be postponed $\#post\_instr_m$ can fit in the available resources of the fetched bundle and its potential additional bundle.

c) If the C2 condition is true: the postponing of the $post\_instr_m$ is allowed, thus the instructions to be postponed are stored to FR for later execution. To do so, the CL configures the EI's switch through the command `ex_schedule_and_store` to pass all the instructions, except the ones stored back in FR. The CL also configures the CI's switch, but this time it stores to the CR the results of the instructions that their duplicate has not been executed yet, $remain\_instr$. The command `com_schedule_and_store` represents this functionality.

d) If the C2 condition is false: the postponing of the $\#post\_instr_m$ is not allowed, thus a new time slot has to be added. To do that, the CL generates a time slot by stalling the pipeline through the signal `pipeline_stall`.

Then, the CL configures the EI's switch on how to pass the instructions and store the postponed ones to the FR for later execution. The command `ex_schedule_and_store` represents this functionality, like previously. CI's switch is configured through `com_schedule_and_store` to store

to the CR the results whose the duplicated value has not been yet computed, as `com_schedule_and_store` of the previous subsection. Since $\overline{C2}$ is true, the CL passes to the next FSM state, S1, where it configures the EI's and CI's switch on how to pass the stored instructions of FR and CR, respectively. Like in paragraph a, the commands `ex_schedule` and `com_schedule` represent these functionalities, but this time having only instructions of FR and CR as an input.

IV. Experimental Results

To validate the proposed approach, ten basic media benchmarks are used which are extracted from MediaBench [8]. The used platform is the VEX VLIW processor [9] with HP VEX compiler. Two VLIW configurations have been explored, based on realistic commercial VLIWs, e.g. Intel Itanium [10], as following: a) 4-issue width (4 ALUs, 2 Mult, 1 Mem, 1 Br) and b) 8-issue width (8 ALUs, 4 Mult, 2 Mem, 1 Br).

We perform experiments by applying two fault tolerance techniques: duplication and triplication of the instructions. NEDA is compared with a state of the art approach named adaptive duplication with ILP reduction [6], [7]. A simulation tool has been developed to calculate the execution cycles in order to estimate the performance of all approaches, from the extracted traces of the processor execution instruction sequence. An area and power evaluation is under development.

Fig. 6 depicts the execution cycles for the 4-issue configuration and Fig. 7 depicts the execution cycles for 8-issue configuration. The results correspond to: a) the unprotected execution (Normal), b) the state of the art approach that NEDA is compared with, for duplication (DMR) and triplication of instructions (TMR) and c) the proposed approach for duplication (DMRi) and triplication (TMRi). Fig. 8 depicts the reduction in performance degradation of the proposed approach with respect to existing approaches calculated by $\frac{DMR-DMRi}{DMR}$ and $\frac{TMR-TMRi}{TMR}$.

For the 4-issue configuration and DMRi method, we observe a reduction from 2% for *bcnt* benchmark to 16% for *crc* benchmark and for the TMRi a reduction from 2% for *bcnt* benchmark to 13% for *crc* benchmark. The average reduction for the DMRi and TMRi is 6% and 5%, respectively. The *crc* outperforms the rest of the benchmarks because its traces mainly consist of sequences of full slots followed by empty ones. This occurs because *crc* is an algorithm that repeatedly checks sub-results, through single compare instructions after a bunch of calculations and shiftings.

For the 8-issue configuration and TMRi method, we observe a reduction from 1% for *huff_ac_dec* benchmark to 12% for *fft32x32s* benchmark. For the TMRi there is a reduction from 12% for *huff_ac_dec* benchmark to 26% for *bcnt* benchmark, which is the maximum reduction observed. The average reduction for the DMRi and TMRi is 7% and 19%, respectively.

The 8-issue width has generally better results than the 4-issue width for both DMRi and TMRi, except for the DMRi of the *huff_ac_dec* and *crc* benchmarks. The increase of the issue width from 4 to 8 does not imply that the ILP will be

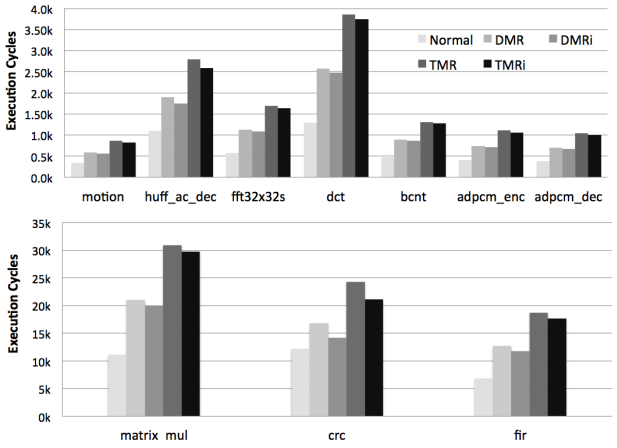


Fig. 6: 4-issue VLIW configuration.

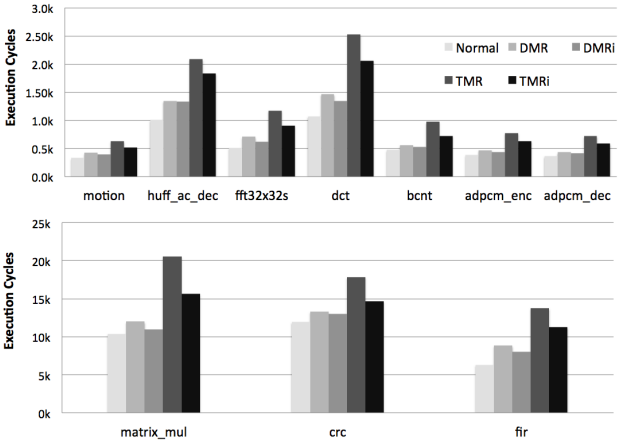


Fig. 7: 8-issue VLIW configuration.

duplicated as well and, thus, more idle slots exist in the 8-issue width. For these applications the ILP is relatively low even for 8-issue width, eliminating the need for extra time slots, since the duplicated instructions can fit in the current bundle. However there is a significant reduction in TMRi , because as the instructions are triplicated they cannot fit in the current bundle, so a new bundle is needed.

We also observe that the reduction of the TMRi outperforms the reduction of DMRi in the 8-issue width. By triplicating the instructions, more time slots may need to be added (up to two for triplication instead of up to one for duplication). As NEDA explores next bundles' idle slots, results become better for TMRi.

For the 4-issue we generally observe similar and relatively lower reduction for both the DMRi and the TMRi. This occurs because of the small issue width and the limitation in terms of FUs. The additional time slots implied by the duplication/triplication techniques cannot always be reclaimed by NEDA, because in many cases the FUs are not as many as needed or because they are not of the preferred type. In contrast with the previous observations, for *crc*, the DMRi outperforms TMRi. This occurs because the duplication of

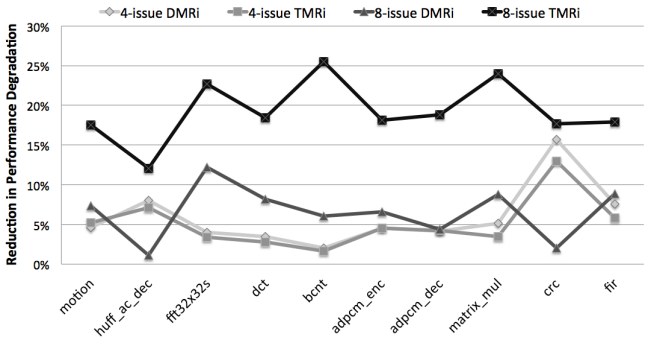


Fig. 8: Reduction in performance degradation

the instructions and the exploration of the idle slots across the instruction bundles creates a quite dense schedule, whereas in the TMRi additional time slots and thus additional idle slots are added where no instruction exist to be executed.

V. Related Work

To provide fault tolerance in processors with inherent resource redundancy, software-based and hardware-based techniques take advantage of the additional resources.

Software-based approaches replicate the instructions, add new comparison instructions and schedule them at design-time. Hence, they can efficiently explore the idle slots to schedule these additional instructions without additional hardware control. However, the code size, the storage needs and the power consumption are increased. For instance, the compiler duplicates the operations and schedules them in different FUs for VLIW processors [11] or exploits the idle slots for soft errors adding a new slot, whenever idle slot exploration is not possible [5]. To reduce the number of additional instructions, software-based approaches are combined with hardware-based ones. The instructions duplication and scheduling is performed by the compiler, whereas the comparison is performed by the hardware. In case of an error, a simple HW operation rebinding re-executes the operation in the next slot [12]. However, the VLIW consists of homogeneous issue slots with FUs executing any type of operations, whereas single errors are considered.

Hardware-based approaches duplicate the instructions at run-time through specific hardware. They eliminating the need of high storage requirements and additional instructions. With respect to statically scheduled data-path, existing approaches maintaint the compiler's result. To do so, one to one coupling of the VLIW pipelines is applied. Therefore, the replicated instructions can also use the schedule given by the compiler for the original instructions [6], [7], [13]. In [13], error detection and mitigation on an 8-issue width VLIW processor is applied. The idle issue slots inside the instruction bundle are used for the execution of the duplicated instructions. If no idle slots exist, the instructions are not duplicated. In [6], [7] the technique is extended with ILP reduction. When a VLIW bundle has more than half of its issue-width filled with instructions, a complete duplication in the same bundle is impossible. The bundle is divided and an additional time slot is added. A trade-off between the number of duplicated instructions and the failure rate is explored.

Other approaches exist, but for dynamically scheduled processors, such as superscalars, where they take advantage of the already existing hardware dynamic scheduler. In [14], the instructions are duplicated in the dynamic scheduler or in the functional units. REMO [15] duplicates instructions in a replay buffer and issues them after the commit of the original instruction. In [16] replay mechanisms are proposed and the redundant instructions are executed in a different function unit.

VI. Conclusion

VLIW processors have hardware redundancy that is not always exploited by application. The remaining idle slots can be used for reliability purposes. Compared with existing approaches, a hardware-based approach is proposed for heterogeneous statically scheduled data paths. NEDA is a run-time idle slot exploitation method with a dependency analysis inside the current instruction bundle, as well as across the next instruction bundle and its potential replication. The instructions are scheduled taking into account the limitations on the number and the type of resources. In this way, an efficient idle slot's utilization is achieved leading to a reduction of performance degradation, up to 26% shown by the experimental results. As no resource coupling is required, NEDA's approach results to a more flexible instruction scheduling.

References

[1] J. W. McPherson, "Reliability challenges for 45nm and beyond," in *DAC*, July 2006, pp. 176–181.
[2] P. Kabisatpathy, A. Barua, and S. Sinha, *Fault Diagnosis of Analog Integrated Circuits*. Springer, 2005.
[3] J. Klecka, W. Bruckert, and R. Jardine, "Error self-checking and recovery using lock-step processor pair architecture," 2002.
[4] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *CGO*, ser. CGO '05. IEEE Computer Society, 2005, pp. 243–254.
[5] J. S. Hu *et al.*, "Compiler-directed instruction duplication for soft error detection," in *DATE*, March 2005.
[6] A. L. Sartor *et al.*, "Adaptive ilp control to increase fault tolerance for vliw processors," in *ASAP*, July 2016, pp. 9–16.
[7] ——, "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] C. Lee *et al.*, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO*, Dec 1997, pp. 330–335.
[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.
[10] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *MICRO*, vol. 20, no. 5, pp. 24–43, 2000.
[11] C. Bolchini and F. Salice, "A software methodology for detecting hardware faults in vliw data paths," in *DFT*, 2001, pp. 170–175.
[12] M. Schözel, "Hw/sw co-detection of transient and permanent faults with fast recovery in statically scheduled data paths," in *DATE*, March 2010, pp. 723–728.
[13] A. L. Sartor *et al.*, "A novel phase-based low overhead fault tolerance approach for vliw processors," in *ISVLSI*, July 2015, pp. 485–490.
[14] M. Franklin, "A study of time redundant fault tolerance techniques for superscalar processors," in *DFT*, Nov 1995, pp. 207–215.
[15] S. Gopalakrishnan and V. Singh, "Remo: Redundant execution with minimum area, power, performance overhead fault tolerant architecture," in *IOLTS*, July 2016, pp. 109–114.
[16] R. Rodrigues, A. Annamalai, and S. Kundu, "A low-power instruction replay mechanism for design of resilient microprocessors," *TECS*, vol. 13, no. 4, pp. 85:1–85:23, Mar. 2014.