Vol.2011-ARC-194 No.9
2011/3/11

Exploiting Efficiency of Redundant Executions on an FU Array

Jun Yao[†1] and Yasuhiko Nakashima[†1]

We introduce an error tolerable processor to perform duplicated executions by supporting an Explicitly REdundant VLIW Architecture (EReLA). EReLA extends the conventional VLIW ISA and provides special data sanity check (DSC) instructions to help compilers insert fail-safe mechanisms into binaries. As the redundant execution will lengthen the data path, we further study a scheme to employ a functional unit (FU) array based accelerator to sufficiently cover the possible performance impact. Our results show that the explicitly denoted fail-safe mechanisms in EReLA can work best with the FU array to tolerate both soft and hard errors. By properly mapping extended binaries onto the FU array, the processor can maintain its loop iteration-based throughput, which indicates a negligible performance cost.

1. Introduction

Recently, the advancing trend in process technology towards ultimate miniaturization has lead to a continuously increasing transient and permanent failure rate in electronic devices including microprocessors. Electronic units which are vulnerable due to their small size, supply voltage and capacitance becomes unreliable to generate correct results. It thereby necessitates the needs for device, architecture and system level mechanisms to help construct fault tolerable processors for a future technology.

In this paper, we propose an architecture to effectively perform fully redundant executions. The baseline architecture was originally constructed to achieve a significant execution speed-up by mapping the hottest spot in a program—which is usually the inner loop—onto a functional unit (FU) array. The data-flow graph in the kernel of that mapped loop is executed along the depth of the FU array, which guarantees finishing the loop iterations in continuous cycles. With minor changes to guarantee error detection and recovery, it is possible to use the FU array to tolerate both transient and permanent failures, and cover possible performance drawbacks from redundant execution by its high execution throughput.

For this purpose, we proposed Explicitly REdundant VLIW Architecture (EReLA), extending a conventional VLIW ISA with necessary augmentation to aid the code replication and the data sanity checking (DSC). Specifically, DSC instructions have been included in EReLA to explicitly indicate the locations and sources to perform error checking. The mapping scheme interprets the modified binaries and maps the lengthened data-flow graph in a highly compacted fashion onto the FU array. In an FU array with a sufficient depth to hold the replicated data-flow graphs, parallelism between different loop iterations can still be fully exploited. This will bring almost no performance impact even with the full redundancy. In addition, EReLA is also expected to cooperate with the scheme to locate permanent defects effectively. The array structure makes it possible to perform a very fine grained unit replacement, without lower its original resource availability.

The paper is organized as follows: Section 2 introduces many architecture level redundant executions as related technologies. Section 3 gives an FU array based accelerator—Linear Array Pipeline Processor (LAPP), as our baseline platform. In Section 4, the proposed EReLA structure and working schemes will be demonstrated in detail. Section 5 concludes the whole paper.

2. Related Technologies

Essentially, electronic faults in microprocessors can be covered by certain levels of redundancy[1)–5)]. As an example, IBM z990[1)] uses a redundant processor core to perform duplicated instructions on-the-fly. Data check is done before each committing stage, either to the register file or data cache. AR-SMT[2)] employs a multi-threading architecture to work on thread level duplication. Qureshi et al[5)] proposes to issue a second execution under the long delay of L2 cache misses, which can thus largely cover performance losses in memory intensive programs.

These space or time redundancies or their combinations will require extra copy of hardware or introduce additional execution delay. Although it is possible to distribute redundant executions onto different cores or time slots in a chip multi

†1 Nara Institute of Science and Technology

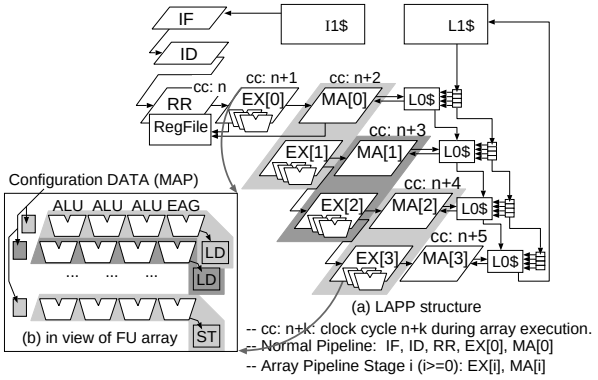

Fig. 1 Structure of LAPP.

processor (CMP) to exploit parallelism and efficiency[6], the access of shared checking data will still affect the performance. Moreover, core-level redundancy will indicate a large granularity of replacement after permanent defects. Disabling a corrupted unit will sometimes cause a decrease in the core number.

Compared to many core or thread level redundancies, current reconfigurable architecture is expected to have potentials in duplicated executions. Reconfigurable architecture is originally designed for better power/performance. It usually contains a large functional unit (FU) array. By properly setting the connections and operations in these FUs, it is possible to work on a certain data-path with a high speed. Accordingly, the large FU pool provides an effective option to exploit parallelism in fully redundant codes. Moreover, after permanent defects, only certain blocks in the large FU array will be disabled by simple isolation, which indicates a very fine granularity replacement. Taking all these features into account, this research is focused on finding an effective working scheme of redundant executions in an FU array based architecture.

3. Linear Array Pipeline Processor

In this section, the structure of Linear Array Pipeline Processor (LAPP) will be introduced. LAPP is a specific accelerator implementation of a reconfigurable FU array based processor for high-speed execution. Besides its high performance feature, LAPP maintains backward compatibility by supporting conventional VLIW which is usually unavailable in a reconfigurable architecture. According to these features, it is selected as the baseline platform for the redundant execution in this research.

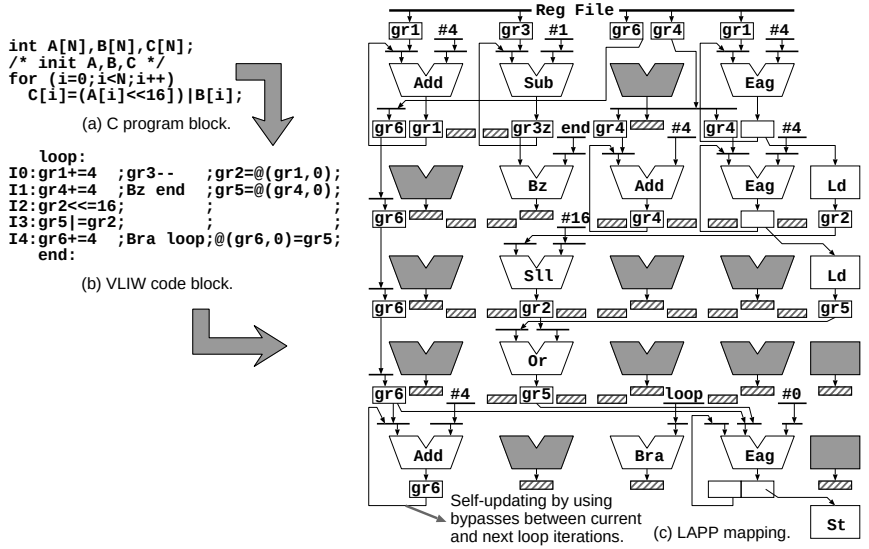

Fig. 2 Mapping a loop kernel onto LAPP.

Fig. 1 shows the basic structure of LAPP. Basically, LAPP contains a normal VLIW pipeline, shown as IF, ID, RR, EX[0], and MA[0] in Fig. 1(a). Additionally, LAPP extends its EX and MA stages into extra working stages, as EX[i] and MA[i] ($i > 0$). The combination of these EX[j]s and MA[j]s ($j \geq 0$) take an FU array format, as depicted in Fig. 1(b). The FU array can be regarded as a series of array pipeline stages, in which each array pipeline stage represents EX[i] and MA[i] in Fig. 1(a).

By properly setting the configuration data, which is the mapping information in each array pipeline stage, it is possible to use LAPP to accelerate the hottest loop (**Fig. 2**(a)) inside the program. Specifically, the program block in Fig. 2(b) can fit into the FU array, taking the mapping of Fig. 2(c).

As shown in Fig. 2(c), the loop kernel can be mapped onto 5 array pipeline stages. The FU array can overlap the executions of different loop iterations, following a pipeline concept. After the first 6 cycles, one loop iteration can be finished per cycle[⋆1], which indicates a significant speed-up by exploiting sufficient parallelism between iterations.

4. Fault Toleration Scheme in EReLA

4.1 Explicit Redundancy by software/hardware approaches

As described in Section 2, electric faults can be tolerated by duplicating original executions either with multiple resources or along the time line. In this research, we use LAPP's large FU resource and its high throughput to aid the redundant execution.

The basic idea of redundant execution in this research works in the following way. Taking the calculation of `S=A+B+C;` as an example, the redundant execution can be achieve by explicitly assuming a new series of program codes as `S1=A+B+C; S2=A+B+C; S=(S1==S2)?S1:recover();`, where `S2` is the secondary calculation. If the calculation of `S1` and `S2` can be distributed to different hardware resources, the comparison after the calculation can help detect the erroneous state in either of the summaries. The procedure of `recover()` can perform re-execution to tolerate transient errors or use a higher dependable architecture like triple modular redundancy (TMR) with voting logic to locate the permanently defected units.

The new program will extend the length of data-flow graph. However, with LAPP, an extended data-flow graph only means that the mapping will require more array pipeline stages. Usually, LAPP prepares a large depth of array pipeline to be ready for mapping loops with a very long data path. Furthermore, some augmentations can be added to re-use mapped array stages in different time slots, which virtually extends the depth of LAPP's array pipeline.

Many different approaches can be employed to translate VLIW codes according to the basic idea above, either by the extended compiler or by dynamic duplication during the LAPP mapping scheme. In this research, we use a compiler-aided approach to explicitly help LAPP better understand the locations and sources of data sanity checks. The extended VLIW code is listed in **Fig. 3**(a).

To support explicit redundancy and data sanity check (DSC), special instructions have been added in VLIW architecture, as `DSC` (1 input), `DSCz` (further covering `z` bit), `DSCadr` (checking address). Every register in LAPP architecture will be extended to contain an additional `f` bit, storing the check result of `DSC`. `DSC` instruction is mainly comparison. `CHK-f&` instructions will invoke `recover` procedure or perform its latter half according to the `f` bit. `pre-ST` instruction calculates the address without taking the real store phase. Only when both address and data to store are checked by `CHK-f&ST`, the data will be committed into the cache. Alternatively, we can also add special treatment in L0 to L1 cache committing to avoid the output of tainted value.

A hardware approach is needed to duplicate instruction except the `DSC` and `CHK-f&` instructions. Fig. 3(b) shows the extended version of Fig. 3(a)[⋆2]. Instructions are now with a primary/secondary program counter (PC) format, taking the style of `I`x`_`y. The data of `I`x`_`$*$ depends on `I`$(x-1)$. Note that it is still possible to use `I`x`_0` to represent the original program in Fig. 3(a). Additionally, as shown in Fig. 3(a), with a good understanding of the program block, it is possible to only put `DSC`s to the data affecting the control path or outputs. This helps to reduce the additional cost in a redundant execution.

All instructions in Fig. 3(b) can be mapped onto the FU array with minor extension of the supported functions. In this paper, we provide a hand-mapping result in Fig. 3(c), which gives the mapping of redundant VLIW loop kernel in Fig. 3(c). Note that all register data are fetched from the register file and propagated forward. Accordingly, the registers between different array stages are required to use hardening technologies like ECC. As demonstrated in Fig. 3(c), the extended kernel can still be mapped within a sufficiently deep array pipeline. In this example, the 10 stages can finish loop iterations in continuous cycles. The performance impact from redundancy will be kept in a very low level when the code loops around sufficient times.

⋆1 There are some restrictions such as no data dependences between different loop iterations except loop counters, and so on. Paper 7) has a clear definition of LAPP working situation.

⋆2 The target of branch is supposed to be calculated during the compiling phase so that instructions like `Bra` is not needed to be duplicated. For high reliability, the control path inside LAPP will be verified by dual hardware resources.

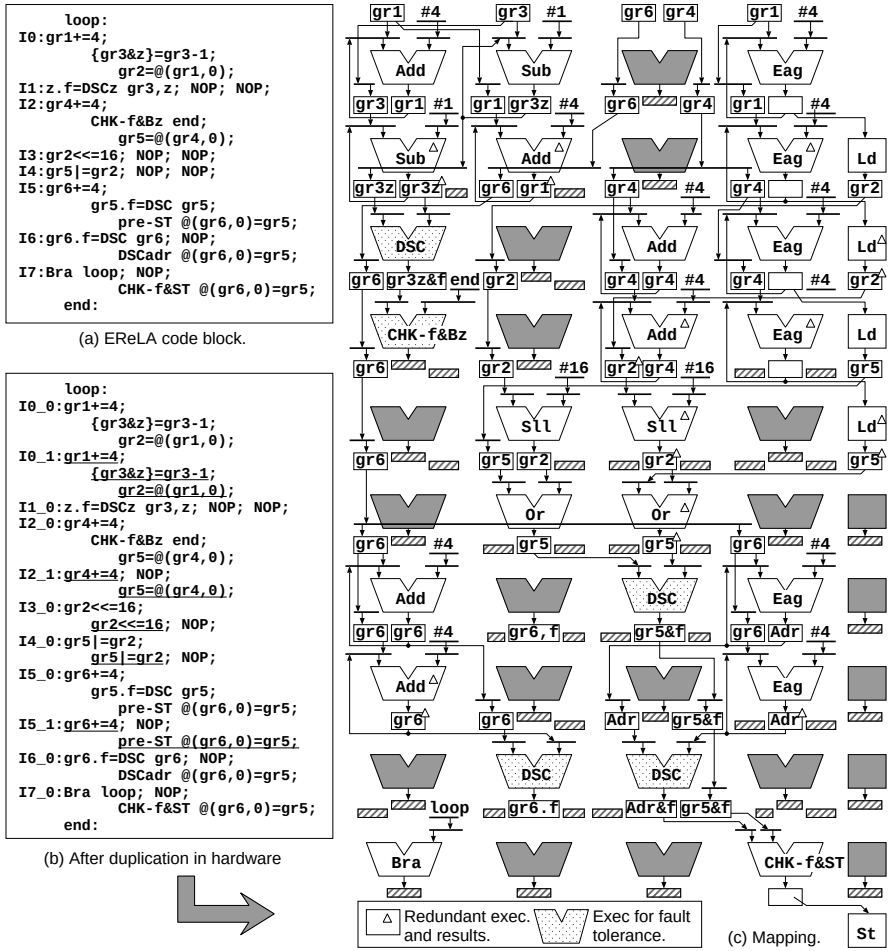

Fig. 3 Mapping EReLA codes onto LAPP.

LAPP uses its normal mode, which is a VLIW processor to execute codes outside a loop kernel or unmappable loops[7]. Similarly to a traditional multi-modular redundant architecture, we can add two normal VLIW cores inside LAPP to perform non-FU array duplicated executions. Additionally, the resources in the FU array, both the network and the FUs can be used for the additional comparison purpose. **Fig. 4** gives a preliminary diagram for this idea. Detailed hardware connection is still under research.

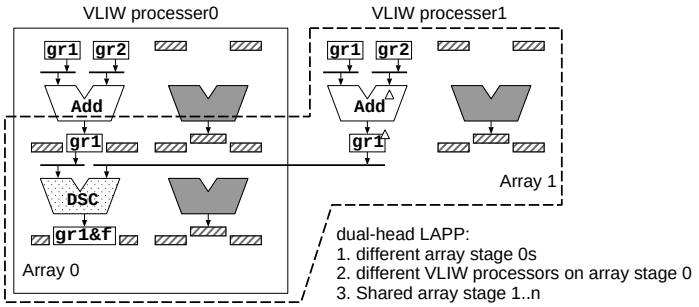

Fig. 4 A dual-head LAPP to cover errors under normal execution.

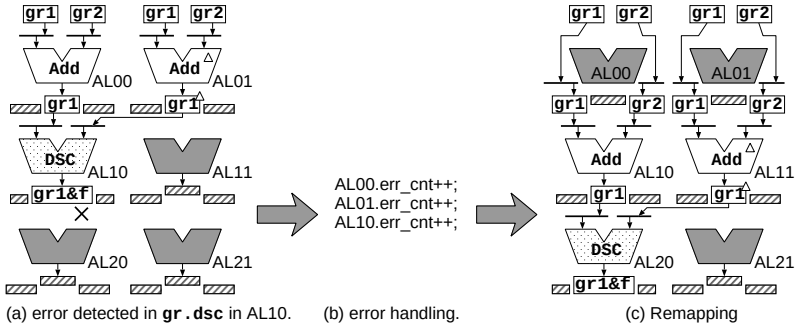

Fig. 5 Using error counter and re-mapping to locate hard error in LAPP.

4.2 Scheme to Locate Permanently Defected Units

The architecture in Fig. 3 can sufficiently detect error during the real execution. When the error is transient, it can be covered by a re-execution based fail-safe procedure. However, when permanent error—also known as hard error—attacks, the defected unit will always generate incorrect results and simple re-executions will continue to fail. Since the `DSC` instruction only takes two inputs, it can not specify the erroneous data path. For this purpose, we adopt the following scheme to tolerate hard errors.

As shown in **Fig. 5**, the FU array redundantly executes `gr1+=gr2` and compares the two results. Suppose `DSC` detects error and set `f` flag to represent this erroneous state. The `recover()` procedure will first increase the `err_cnt` fields in all the mapped units. In this example, `err_cnt` fields of `AL00`, `AL01`, `AL10` will be incremented. In the next mapping phase, the unused `AL2X` array pipeline stage will be included in the mapping as shown in Fig. 5(c). Eventually, the `err_cnt` bits of certain ALUs will reach some pre-determined threshold, which indicates a location of permanent defected states. Accordingly, a free array pipeline stage will always be provided inside the array The free stage will rotate along the array depth, during the remapping phase.

5. Conclusions

In this paper, we give an FU array based accelerator architecture, which works on VLIW codes with explicit redundancy and data checks. An extended VLIW ISA with the name of EReLA is proposed for the purpose of generating high reliable binaries. The proposal provides an optimized interface between the FU array based processor and the program binaries to tolerate both soft and hard errors. Our mapping result in this paper shows that the software/hardware approaches can sufficiently maintain processor throughputs and conceal performance impact from redundant executions.

Acknowledgments This work is supported by JST ALCA (Advanced Low Carbon Technology Research and Development) Program. This work is partially supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration with Synopsys Corporation.

References

1) Meaney, P., Swaney, S., Sanda, P. and Spainhower, L.: IBM z990 Soft Error Detection and Recovery, *Device and Materials Reliability, IEEE Transactions on*, Vol.5, No.3, pp.419–427 (2005).
2) Rotenberg, E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors, *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pp.84–91 (1999).
3) Oh, N., Shirvani, P. and McCluskey, E.: Error Detection by Duplicated Instructions in Super-Scalar Processors, *IEEE Transactions on Reliability*, Vol.51, No.1, pp.63–75 (2002).
4) Reinhardt, S.K. and Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading, *Proceedings of the 27th annual international symposium on Computer architecture*, pp.25–36 (2000).
5) Qureshi, M.K., Mutlu, O. and Patt, Y.N.: Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors, *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp.434–443 (2005).
6) Gomaa, M., Scarbrough, C., Vijaykumar, T.N. and Pomeranz, I.: Transient-Fault Recovery for Chip Multiprocessors, *Proceedings of the 30th annual international symposium on Computer architecture*, pp.98–109 (2003).
7) Yoshimura, K., Iwakami, T., Nakada, T., Yao, J., Shimada, H. and Nakashima, Y.: An Instruction Mapping Scheme for FU Array Accelerator, *IEICE Transactions on Information and Systems*, Vol.E94-D, No.2, pp.286–297 (2011).