Compiler-Directed Instruction Duplication for Soft Error Detection

Jie S. Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan, Mary J. Irwin

Compiler-Directed Instruction Duplication for Soft Error Detection*

Jie S. Hu[1], Feihui Li[2], Vijay Degalahal[2], Mahmut Kandemir[2], N. Vijaykrishnan[2], Mary J. Irwin[2]
[1]Department of Electrical and Computer Engineering, New Jersey Institute of Technology
[2]Department of Computer Science and Engineering, The Pennsylvania State University

Abstract

In this work, we experiment with complier-directed instruction duplication to detect soft errors in VLIW datapaths . In the proposed approach, the compiler determines the instruction schedule by balancing the permissible performance degradation with the required degree of duplication. Our experimental results show that our algorithms allow the designer to perform tradeoff analysis between performance and reliability.

1. Introduction

Transient faults due to soft errors are becoming an important concern in current computing systems. Consequently, techniques to improve the reliability of application execution in VLIW processors are of interest. In this work, we focus on detecting the transient errors in the VLIW datapath through complier directed instruction duplication. Since the compiler has full control of instruction scheduling in VLIW architectures, the duplication is controlled at compilation time. This has a distinct advantage over runtime techniques as the compiler can balance power, performance, and code size requirements when performing the duplication. Prior work [2, 1] on soft error detection in VLIW architectures fully duplicates instructions and includes additional comparison instructions to check the results of the duplicate executions. Our experiments reveal that full duplication can be costly from both performance and energy perspectives. In fact, while full duplication is essential in safety-critical applications, for large segments of embedded markets, factors such as power and performance will continue to be as important as reliability.

2. Compiler-Directed Instruction Duplication

The main idea behind our algorithm is to fill empty execution slots (cells) with duplicate instructions under a performance bound. The instruction duplication mechanism in our approach is supported by an hardware enhancement for efficient result comparison that significantly improves our full-duplication scheme with a performance loss of 10% compared to 70.9% reported in [1]. We implemented our instruction duplication approach within the Trimaran framework, which is an integrated compiler/simulation environment. Our results clearly show that our algorithms can be used for improving reliability against soft errors under performance bounds. The following subsections present the architecture and compiler support needed to implement the scheme.

* This work was supported by NSF grant no. 0093082, 0093085 and 0202007

2.1. Architectural Support

Architectural support is required for associating a particular instruction with its duplicate. We maintain a integer/floating-point register value queue (IRVQ/FRVQ) and a load/store address queue (LSAQ) for verifying the outputs of instructions (see Figure 1). When a register-output instruction (generates register value) completes, it writes the value into the output register as well as in the RVQ. When the duplicate of this instruction completes its execution, its results are compared with the contents of the RVQ entry associated with the original. A single bit (B1) is used for identifying the instruction as either original (1) or duplicate (0), and the ordering requirement is achieved by appropriate labeling of this bit based on the relative ordering between the original and the duplicate in the instruction schedule. There is an additional bit (B2) associated with each instruction indicating the required action to the RVQ. If the original instruction precedes its duplicate, B2 bit of the original instruction is set. In the duplicate instruction, B2 is also set and B1 is reset to 0. After finishing execution, the original instruction writes to the tail slot of the RVQ, and the duplicate compares its output with this corresponding entry in the RVQ. If both the original and duplicate instructions are scheduled at the same cycle, both B2 bits are reset. These two instructions compare their outputs immediately after execution without invoking any Write/Read to the RVQ.

2.2. Compiler Support

We see the compiler-scheduled code as a two-dimensional table, where the rows correspond to schedule cycles and the columns represent functional units. Each entry in this table is referred to as a *cell.* An empty *cell* represents an idle cycle. *Duplication range* of an instruction i is the range of cycles within which a duplicate of the instruction can be scheduled. This range is determined by the instructions that i depends on as well as the instructions that overwrite the register read by i. More specifically, we cannot schedule the duplicate before the source operands for the instruction are ready. Similarly, we need to schedule the duplicate before any of the source operands are overwritten (see Figure 2).

2.2.1. Duplication under Performance Bound For duplicating under the presence of a performance bound, given that the original (i.e., without any duplicates) schedule length is C cycles, our objective is to maximize the number of instructions with duplicates so that the new schedule length, C', is smaller than or equal to $(1 + f) \times C$, where f is an input parameter, set by the user, that enforces the tolerable limit in performance degradation. One can conduct a reliability-performance tradeoff analysis by varying the value of f. Basically, this algorithm considers each instruction in turn, identifies its duplication range, and cre-

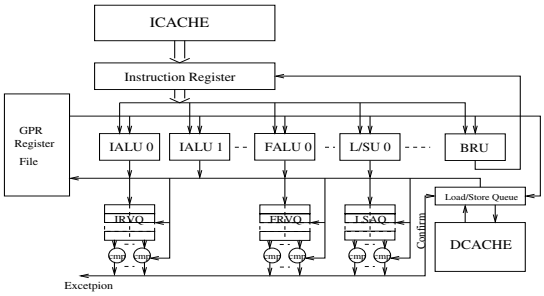

Fig 1: Architectural support

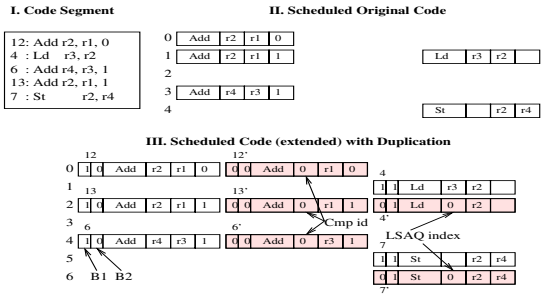

Fig 2: An example code (shaded instructions are duplicates)

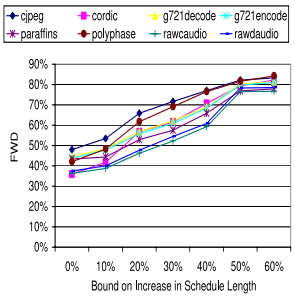

Fig 3: FWD values

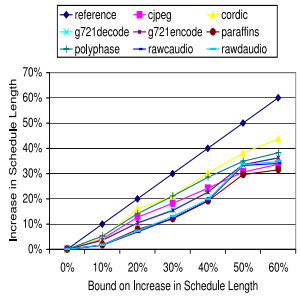

Fig 4: Increase in the original schedule length

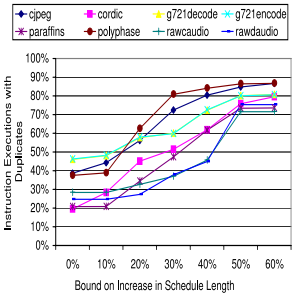

Fig 5: Percentage instruction executions with duplicates

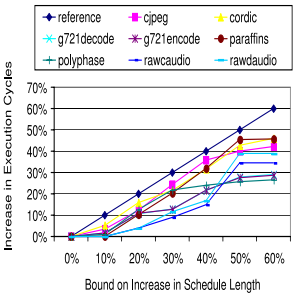

Fig 6: Increase in the original execution cycles

ates a duplicate for it if doing so does not cause the schedule length to exceed $(1 + f) \times C$.

3. Results

We evaluate the proposed technique using both compile-time and run-time metrics. The compile-time reliability metric is the *fraction of instructions with duplicates,* or FWD for short. Intuitively, more duplication should result in more reliable execution assuming a uniform distribution of potential soft errors. The run-time reliability metric, on the other hand, is the fraction of instructions executed with their duplicates. As for the performance metric, the compile-time one is the percentage increase in the static schedule length, and the run-time one is the percentage increase in the program execution time with respect to the case without any duplication. Throughout our discussion we use the term *original schedule* to refer to the version without any duplication.

3.1. Compile-Time Evaluation

Our experimental results show that the average FWD values across our benchmarks are 86.8% and 41.4% for the full-duplication scheme and the scheme with zero performance penalty. The encouraging news is that our approach is able to duplicate more than 40% of the instructions without an increase in the original schedule cycles. Note that, the full-duplication scheme also incurs an average increase of 42.2% in the original schedule length. These FWD and schedule length results clearly illustrate the tradeoffs between reliability and performance. Figure 3 shows the FWD values under different bounds on the percentage increase in the original schedule length. For example, a point $x\%$ on the x-axis indicates that we can tolerate at most $x\%$ increase in the original schedule length. We see that, as expected, the FWD value increases as we relax the performance bound. The curves in Figure 4 shows the actual increase in the original schedule length. The curve labeled "reference" represents the worst case scenario where the schedule for each basic block is increased by $x\%$. We see that the actual schedule length increases are always lower than the maximum allowable limit.

3.2. Run-Time Evaluation

Our experimental results show that the runtime behavior matches very well the static behavior extracted by the compiler. Figure 5 gives the percentage of instructions that are executed with their duplicates under different maximum allowable increases in the original schedule length. We see that these values are similar to the corresponding values given in Figure 3 and 4, indicating that the compile-time reliability estimate is reasonable. The graph in Figure 6 shows the percentage increase in the original execution cycles. Note that these values are lower than the corresponding bounds for all benchmarks.

4. Concluding Remarks

In this paper, we present a study of instruction duplication for detecting soft errors in VLIW datapaths and present compiler algorithms for this purpose. The characteristic of the algorithm is that it improves reliability under performance constraints. Overall, this paper shows that our algorithm allows the designer to conduct tradeoff analyses between performance and reliability.

References

[1] C. Bolchini and F. Salice. A software methodology for detecting hardware faults in vliw data path. In *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001.

[2] J. Holm and P. Banerjee. Low cost concurrent error detection in a vliw architecture using replicated instructions. In *Proc. International Conference on Parallel Processing*, 1992.